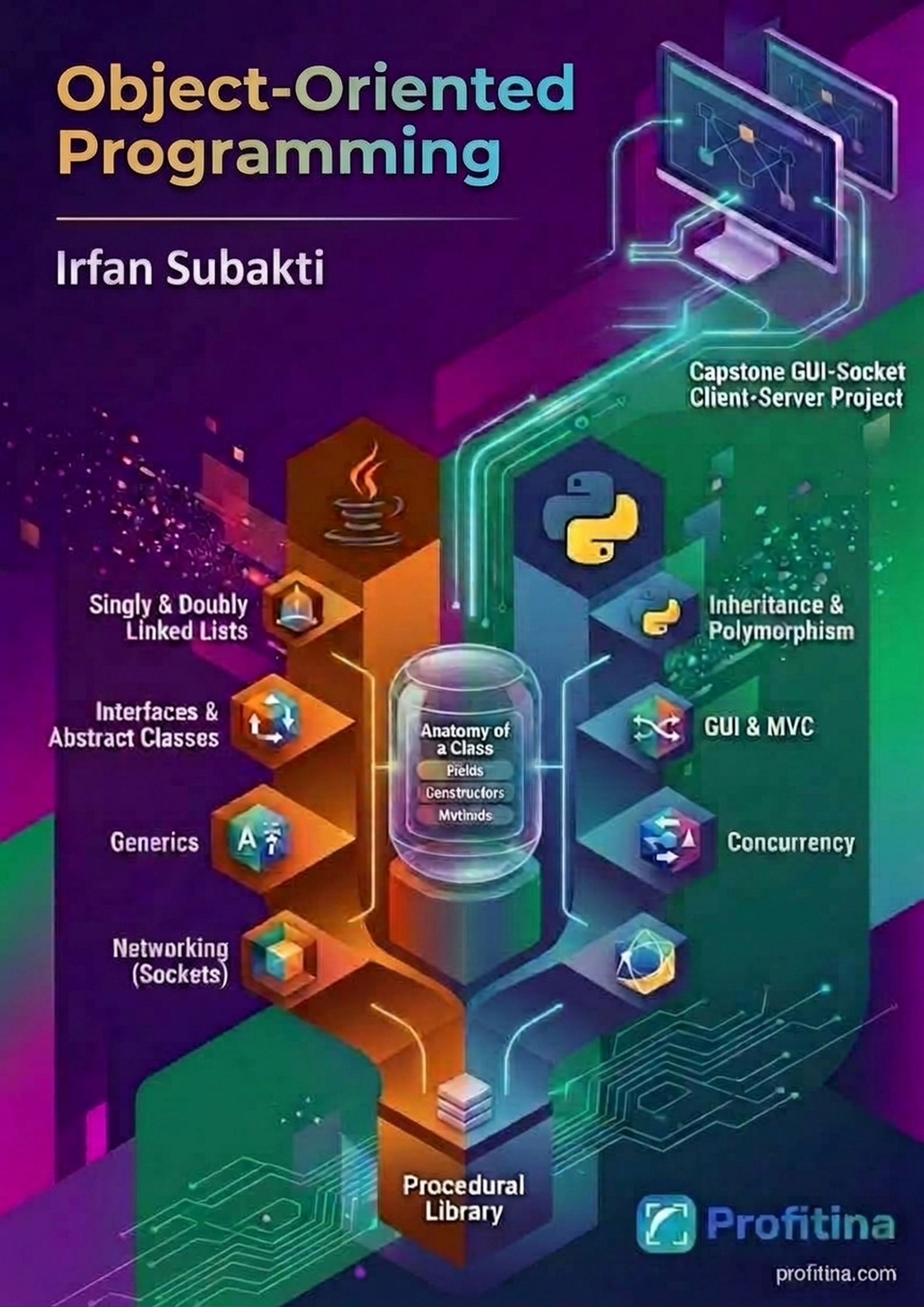
Object-Oriented Programming

Irfan Subakti



Object-Oriented Programming

with Java and Python

Irfan Subakti



Copyright © 2026 Irfan Subakti. All rights reserved.

Published by Profitina · profitina.com

First edition, September 2026.

Every program in this book was compiled and run before its output was printed. The complete source code is in the accompanying Code folder. The programs are offered for teaching and learning; Irfan Subakti and the publisher accept no liability for their use elsewhere.

Contents

Preface ..... **liv**
Who this book is for liv
Why two languages liv
What is in it liv
How to read it lv
About the author lv
Acknowledgments lv

CHAPTER 1 Objects, Programs, and Your First Workbench **1**
1.1 Where We Begin 1
1.2 What a Program Is 2
1.2.1 A recipe for the counter 2
1.2.2 Languages for people, languages for machines 3
1.2.3 From source code to a running program 3
1.2.4 Data and the operations on it 4
1.3 Thinking in Objects 5
1.3.1 Looking around the shop 5
1.3.2 State, behavior and identity 5
1.3.3 Classes: the blank form 6
1.3.4 Messages: how objects work together 7
1.3.5 Abstraction: keeping only what matters 7
1.3.6 The four big ideas, previewed 8
1.4 From Procedures to Objects 8
1.4.1 The notebook program 8
1.4.2 What went wrong 10
1.4.3 The object answer 10
1.5 Why Java, and Why Python 11
1.5.1 Java: a short history 11
1.5.2 Why Java: portable, supported, everywhere 12
1.5.3 Python: a short history, and why Python too 12
1.5.4 What the popularity rankings say 13
1.5.5 The two languages compared 15
1.5.6 One way of thinking, two spellings 16
1.6 The Tools of the Trade 16
1.6.1 The JDK and the JRE 16
1.6.2 Inside the JDK: the `bin` folder and the PATH 17
1.6.3 The Python interpreter and its companions 18
1.6.4 Integrated development environments 18
1.7 Setting Up VS Code for Java and Python on Windows, macOS, and Linux 19
1.7.1 Windows, step by step 20
1.7.2 macOS, step by step 21
1.7.3 Linux, step by step 22
1.7.4 VS Code and its two extensions 22
1.7.5 Your first Java program 23
1.7.6 Your first Python program 25
1.7.7 Debugging: watching a program think 26
1.7.8 When something goes wrong 28
1.8 Your First Class: a Laundry Order 28
1.8.1 A receipt: variables and arithmetic 28
1.8.2 Designing the `Order` class 30

1.8.3 The `Order` class in Java 30
1.8.4 The `Order` class in Python 32
1.8.5 Bringing orders to life 34
1.8.6 What a variable of a class type holds 35
1.8.7 What the class has bought us 36
1.9 The Road Map of This Book 36
1.10 Profitina in Practice 37
1.10.1 The worked case study: a day at Profitina Laundry 38
1.11 Common Mistakes, and How to Avoid Them 41
1.12 Chapter Summary 42
1.13 Exercises 42
1.14 Solutions to the Exercises 43
References 47

CHAPTER 2 The Development Environment: Editing, Running, and Debugging 50
2.1 Where We Left Off 50
2.2 What an IDE Is, and Why You Want One 51
2.2.1 A workbench, not a magic box 51
2.2.2 The five jobs of an IDE 51
2.2.3 Eclipse and VS Code: two workbenches, one set of ideas 51
2.3 How a Program Runs: Compiling, Interpreting, and Virtual Machines 52
2.3.1 The processor speaks only one language 52
2.3.2 Compilers and interpreters, and when each complains 53
2.3.3 Java: javac, bytecode, and the Java Virtual Machine 53
2.3.4 Python: the interpreter, its hidden compiler, and `__pycache__` 55
2.3.5 When mistakes are caught: compile time and run time 56
2.4 Installing and Starting Eclipse 57
2.4.1 Downloading and installing 57
2.4.2 The workspace 58
2.4.3 Appearance 59
2.5 The Eclipse Workbench: Views, Editors, and Perspectives 60
2.5.1 Views and editors 60
2.5.2 Perspectives 61
2.5.3 The views of the Java perspective 63
2.5.4 Projects and their natures 65
2.6 Your First Eclipse Project, Package, and Class 65
2.6.1 Creating the project 65
2.6.2 Creating a package 66
2.6.3 Creating a class and running it 67
2.6.4 Running the program outside Eclipse: the JAR file 70
2.7 The Same Work in VS Code 72
2.7.1 Folders instead of workspaces 72
2.7.2 Java in VS Code: the Java Projects view 72
2.7.3 Python in VS Code: choosing the interpreter 73
2.7.4 Run configurations: `launch.json` 73
2.7.5 The terminal is always there 74
2.8 Projects, Packages, Modules, and Imports 74
2.8.1 Why group code at all 74
2.8.2 Packages are folders 75
2.8.3 A project with two packages 75
2.8.4 The missing import, in both languages 79
2.8.5 Compiling a project with packages by hand 80
2.8.6 Exporting, importing, and navigating projects 81
2.9 Editing Power Tools: Content Assist, Quick Fix, Refactoring, and Formatting 83
2.10 Debugging from Zero 84

2.10.1 What a bug is, and two ways to look for one ... 84
2.10.2 The vocabulary ... 84
2.10.3 Breakpoints and starting the debugger in Eclipse ... 84
2.10.4 Controlling execution: resume, step over, step into, step out ... 87
2.10.5 The call stack ... 88
2.10.6 Looking at variables, and changing them ... 89
2.10.7 Managing breakpoints: disabling, skipping, hit counts, and conditions ... 90
2.10.8 The same debugger in VS Code, for Java and for Python ... 91
2.11 A Worked Hunt: The Laundry Bill That Was Too High ... 92
2.11.1 The symptom ... 92
2.11.2 Step 1: know what the answer should be ... 95
2.11.3 Step 2: form a hypothesis, then place a breakpoint ... 95
2.11.4 Step 3: a guided tour with step into, step over and step out ... 95
2.11.5 Step 4: watch the running total ... 99
2.11.6 Step 5: fix, and check ... 102
2.11.7 The method behind the hunt ... 105
2.12 Conditional Breakpoints: Finding the Needle in the Haystack ... 105
2.13 Reading Stack Traces ... 109
2.13.1 Reading the Java stack trace ... 110
2.13.2 Reading the Python traceback ... 111
2.14 Profitina in Practice ... 111
2.15 Common Mistakes, and How to Avoid Them ... 112
2.16 Chapter Summary ... 113
2.17 Exercises ... 113
2.18 Solutions to the Exercises ... 114
References ... 119

CHAPTER 3 Classes, Objects, Strings, Types, Conditionals, and Loops ... 120
3.1 Where We Left Off ... 120
3.2 A First Look Inside a Class ... 120
3.2.1 Hello, once more ... 120
3.2.2 Reading words from the command line ... 122
3.2.3 Names and conventions ... 123
3.2.4 Comments that help the reader ... 124
3.3 Classes and Objects ... 125
3.3.1 Blueprints and houses ... 125
3.3.2 The anatomy of a class ... 126
3.3.3 The word `this`, and the word `self` ... 130
3.3.4 Getters, setters, and `toString` ... 131
3.3.5 Creating objects with `new` ... 131
3.3.6 Checking that a class works ... 133
3.4 References and Memory Pictures ... 136
3.4.1 Values and arrows ... 136
3.4.2 Python: every variable is a name ... 139
3.4.3 How to draw an object diagram ... 140
3.5 Strings ... 141
3.5.1 Text is an object ... 141
3.5.2 Positions and the everyday methods ... 141
3.5.3 Strings never change ... 144
3.5.4 Formatting output ... 147
3.5.5 From text to numbers, and back ... 148
3.6 Types ... 150
3.6.1 Java's eight primitive types ... 150
3.6.2 Static typing and dynamic typing ... 151
3.6.3 Characters and truth values ... 152

3.7 Arithmetic and Its Pitfalls 154
3.7.1 Operators and precedence 154
3.7.2 Pitfall one: integer overflow 155
3.7.3 Pitfall two: the computer cannot count to one in tenths 157
3.7.4 Pitfall three: integer division 159
3.8 Conditionals 162
3.8.1 Doing something only when a condition holds 162
3.8.2 Chains of decisions 163
3.8.3 `switch`: choosing among many values 164
3.9 Loops 167
3.9.1 `while` and `for` 167
3.9.2 Four sequences 170
3.9.3 The for-each loop 172
3.9.4 Choosing the right loop 173
3.10 Tracing Tables 173
3.11 Worked Exercises, Step by Step 175
3.11.1 The area of a circle 175
3.11.2 From imperial units to kilograms 176
3.11.3 An investment over five hundred years 179
3.12 Profitina Laundry: Customer and Service 184
3.12.1 Finding the classes 184
3.12.2 The price rules as formulas 184
3.12.3 The code 185
3.12.4 Checking the prices by hand 192
3.13 Profitina in Practice 193
3.14 Common Mistakes, and How to Avoid Them 193
3.15 Chapter Summary 194
3.16 Exercises 195
3.17 Solutions to the Exercises 196
References 205

CHAPTER 4 Evaluation and Discussion I: From a Plain Specification to a Tested Class 206
4.1 Where We Left Off 206
4.2 How to Approach a Programming Problem 207
4.2.1 The eight steps 207
4.2.2 How the tests in this chapter are written 208
4.2.3 How each problem is laid out 209
4.3 Problem 4.1 — Fraction Calculator 209
4.3.1 The problem 209
4.3.2 Understanding it by hand 210
4.3.3 Classes, fields and methods 210
4.3.4 The Java solution 210
4.3.5 The Python solution 212
4.3.6 Line by line 214
4.3.7 A tempting wrong solution 214
4.4 Problem 4.2 — Analog Clock Angle 215
4.4.1 The problem 215
4.4.2 Understanding it by hand 216
4.4.3 Classes, fields and methods 217
4.4.4 The Java solution 218
4.4.5 The Python solution 219
4.4.6 Line by line 221
4.4.7 A tempting wrong solution 221
4.5 Problem 4.3 — Computer Account 222
4.5.1 The problem 222

4.5.2 Understanding it by hand 223
4.5.3 Classes, fields and methods 223
4.5.4 The Java solution 223
4.5.5 The Python solution 225
4.5.6 Line by line 227
4.5.7 A tempting wrong solution 228
4.6 Problem 4.4 — Complex Numbers 229
4.6.1 The problem 229
4.6.2 Understanding it by hand 229
4.6.3 Classes, fields and methods 229
4.6.4 The Java solution 230
4.6.5 The Python solution 231
4.6.6 Line by line 233
4.6.7 A tempting wrong solution 234
4.7 Problem 4.5 — Weight Conversion 234
4.7.1 The problem 234
4.7.2 Understanding it by hand 235
4.7.3 Classes, fields and methods 235
4.7.4 The Java solution 235
4.7.5 The Python solution 236
4.7.6 Line by line 237
4.7.7 A tempting wrong solution 238
4.8 Problem 4.6 — A Fraction Class 239
4.8.1 The problem 239
4.8.2 Understanding it by hand 239
4.8.3 Classes, fields and methods 239
4.8.4 The Java solution 240
4.8.5 The Python solution 242
4.8.6 Line by line 244
4.8.7 A tempting wrong solution 244
4.9 Problem 4.7 — Player and Enemy 246
4.9.1 The problem 246
4.9.2 Understanding it by hand 246
4.9.3 Classes, fields and methods 247
4.9.4 The Java solution 247
4.9.5 The Python solution 250
4.9.6 Line by line 252
4.9.7 A tempting wrong solution 253
4.10 Problem 4.8 — Money in Rupiah 254
4.10.1 The problem 254
4.10.2 Understanding it by hand 254
4.10.3 Classes, fields and methods 255
4.10.4 The Java solution 255
4.10.5 The Python solution 257
4.10.6 Line by line 259
4.10.7 A tempting wrong solution 259
4.11 Problem 4.9 — Time of Day 260
4.11.1 The problem 260
4.11.2 Understanding it by hand 260
4.11.3 Classes, fields and methods 261
4.11.4 The Java solution 261
4.11.5 The Python solution 263
4.11.6 Line by line 265
4.11.7 A tempting wrong solution 265
4.12 Problem 4.10 — A Laundry Receipt 266

4.12.1 The problem 266
4.12.2 Understanding it by hand 266
4.12.3 Classes, fields and methods 266
4.12.4 The Java solution 267
4.12.5 The Python solution 270
4.12.6 Line by line 272
4.12.7 A tempting wrong solution 273
4.13 Problem 4.11 — Tidying a Name 274
4.13.1 The problem 274
4.13.2 Understanding it by hand 274
4.13.3 Classes, fields and methods 274
4.13.4 The Java solution 275
4.13.5 The Python solution 276
4.13.6 Line by line 278
4.13.7 A tempting wrong solution 278
4.14 Problem 4.12 — Password Rules 280
4.14.1 The problem 280
4.14.2 Understanding it by hand 280
4.14.3 Classes, fields and methods 280
4.14.4 The Java solution 280
4.14.5 The Python solution 282
4.14.6 Line by line 284
4.14.7 A tempting wrong solution 284
4.15 Problem 4.13 — Tracing Loops by Hand 285
4.15.1 The problem 285
4.15.2 Understanding it by hand 285
4.15.3 Classes, fields and methods 286
4.15.4 The Java solution 286
4.15.5 The Python solution 288
4.15.6 Line by line 290
4.15.7 A tempting wrong solution 290
4.16 Problem 4.14 — Leap Years and the Next Day 291
4.16.1 The problem 291
4.16.2 Understanding it by hand 291
4.16.3 Classes, fields and methods 292
4.16.4 The Java solution 292
4.16.5 The Python solution 294
4.16.6 Line by line 296
4.16.7 A tempting wrong solution 296
4.17 The Problems Side by Side 297
4.18 Profitina in Practice 298
4.19 Common Mistakes, and How to Avoid Them 299
4.20 Chapter Summary 300
4.21 Exercises 300
4.22 Solutions to the Exercises 301
References 302

CHAPTER 5 Arrays, Lists, Input, Exceptions, and Files 304

5.1 Where We Left Off 304
5.2 Many Values Under One Name: Arrays 305
5.2.1 The pigeonhole rack 305
5.2.2 Creating, filling, and walking an array 306
5.2.3 Stepping off the end 308
5.2.4 Arrays of numbers, and arrays of arrays 310
5.2.5 An array variable holds a reference 311

5.2.6 The array in `main`: command-line arguments ... 313
5.3 Lists That Grow: `ArrayList` and Python's `list` ... 314
5.3.1 Why a fixed size is not enough ... 314
5.3.2 The everyday operations ... 314
5.3.3 How a list grows ... 317
5.3.4 Lists of numbers: boxing ... 318
5.3.5 Array or list? ... 320
5.4 Reading Input from the Keyboard ... 320
5.4.1 `Scanner` and `input()` ... 320
5.4.2 Reading until a sentinel ... 322
5.5 When Things Go Wrong: Exceptions ... 323
5.5.1 A program meets a user ... 323
5.5.2 Reading a stack trace ... 324
5.5.3 A family tree of failures ... 328
5.5.4 `try` and `catch` ... 329
5.5.5 Several `catch` clauses: progressively bigger nets ... 331
5.5.6 `finally`: the block that always runs ... 333
5.5.7 Throwing an exception yourself ... 336
5.6 Checked and Unchecked Exceptions ... 338
5.6.1 Unchecked exceptions: usually our own mistakes ... 338
5.6.2 Checked exceptions: the compiler insists ... 340
5.6.3 Catch, or declare? ... 342
5.6.4 Python's position ... 342
5.7 Exceptions of Your Own ... 343
5.8 Files: Streams, Readers, and Writers ... 346
5.8.1 Streams and readers ... 346
5.8.2 Closing what you open ... 347
5.8.3 `try`-with-resources and `with` ... 349
5.8.4 Writing and appending text ... 351
5.8.5 CSV files ... 354
5.9 Profitina Laundry: Loading and Saving Orders ... 355
5.9.1 The `Order` class ... 357
5.9.2 The exception ... 359
5.9.3 Loading and saving ... 360
5.9.4 The application ... 364
5.10 Profitina in Practice ... 368
5.11 Common Mistakes, and How to Avoid Them ... 369
5.12 Chapter Summary ... 370
5.13 Exercises ... 371
5.14 Solutions to the Exercises ... 372
Further reading ... 384
References ... 384

CHAPTER 6 Documentation, Testing, and Interfaces ... 386

6.1 Where We Left Off ... 386
6.2 A Class, Looked at Once More ... 387
6.2.1 More than one way to be born: several constructors ... 387
6.3 Documentation Comments: Javadoc and Docstrings ... 391
6.3.1 Javadoc, step by step ... 392
6.3.2 Docstrings, step by step ... 394
6.3.3 Examples that check themselves: doctest ... 394
6.4 Generating and Reading Documentation ... 394
6.4.1 javadoc ... 395
6.4.2 pydoc ... 396
6.4.3 Reading the API: the class `Random` ... 397

6.5 Testing from Zero ... 398
6.5.1 The class under test ... 399
6.5.2 A test with no library at all ... 400
6.6 The Anatomy of a Unit Test ... 403
6.6.1 Comparing numbers with a fractional part ... 403
6.7 JUnit 6 and unittest ... 405
6.7.1 A JUnit 6 test class ... 405
6.7.2 Running JUnit 6 in VS Code and in a terminal ... 407
6.7.3 unittest ... 407
6.7.4 pytest ... 409
6.8 Choosing Test Cases, and Writing Them First ... 409
6.8.1 Equivalence classes and boundaries ... 409
6.8.2 Test first ... 410
6.9 When Are Two Objects Equal? ... 411
6.9.1 Identity and equality ... 411
6.9.2 Writing equals and hashCode properly ... 416
6.10 Describing an Object: toString, str and repr ... 422
6.11 Many Versions of a Method: Overloading ... 422
6.11.1 How Java chooses an overload ... 425
6.11.2 Python has no overloading — and does not need it ... 427
6.12 Interfaces ... 429
6.12.1 Why interfaces matter: a plug-in story ... 434
6.13 Swapping a Representation Behind an Interface ... 434
6.13.1 Without an interface ... 434
6.13.2 With an interface ... 439
6.13.3 Type hierarchies: subtypes and supertypes ... 443
6.13.4 Sharing code: default methods ... 443
6.14 Interfaces You Already Use ... 446
6.14.1 Comparable, and casting down a hierarchy ... 447
6.14.2 Listeners: the idea behind ActionListener ... 454
6.15 Python's Interfaces: Abstract Base Classes and Protocols ... 458
6.16 Programming to an Interface: Pricing at Profitina Laundry ... 458
6.16.1 The order ... 459
6.16.2 The interface ... 463
6.16.3 Three basic rules ... 465
6.16.4 Rules that wrap other rules ... 468
6.16.5 The program that uses the rules ... 471
6.16.6 Testing every rule ... 475
6.16.7 What the interface bought us ... 482
6.17 Profitina in Practice ... 483
6.18 Common Mistakes, and How to Avoid Them ... 483
6.19 Chapter Summary ... 484
6.20 Exercises ... 485
6.21 Solutions to the Exercises ... 487
References ... 505

CHAPTER 7 Inheritance, Polymorphism, and Access Control ... 506
7.1 Where We Left Off ... 506
7.2 The Duplication Problem: Two Databases That Repeat Each Other ... 507
7.2.1 The vocabulary ... 508
7.3 Superclasses and Subclasses in Java and Python ... 508
7.3.1 "It is as if LinearDB were defined as…" ... 511
7.3.2 The same program in Python ... 512
7.4 Redefining Methods: Overriding ... 514
7.4.1 The rules Java checks when you override ... 523

7.5 Constructors and `super` ... 523
7.6 Polymorphism: One Call, Many Behaviors ... 528
7.6.1 Reading the Animal class ... 535
7.6.2 Reading the test program ... 536
7.6.3 Overloading versus overriding ... 536
7.7 How Dynamic Dispatch Really Works: a Picture of Memory ... 541
7.7.1 Two types for every variable ... 542
7.7.2 The memory picture: method tables ... 545
7.7.3 The same idea in Python ... 547
7.8 The Root of Every Hierarchy: `Object` and `object` ... 549
7.9 Abstract Classes ... 553
7.9.1 The Java library is built the same way ... 560
7.10 Inheriting from Concrete Classes, and Why to Be Careful ... 560
7.11 Composition versus Inheritance ... 565
7.11.1 The fragile base class ... 566
7.11.2 When is inheritance right? The substitution principle ... 570
7.12 Multiple Inheritance: Python's MRO and Java's Interfaces ... 570
7.12.1 Java: one superclass, many interfaces ... 570
7.12.2 Python: real multiple inheritance and the MRO ... 573
7.13 Access Control ... 576
7.13.1 Java's four access levels ... 576
7.13.2 Choosing access levels ... 579
7.13.3 Python: conventions instead of rules ... 579
7.14 Encapsulation: Objects That Keep Their Promises ... 581
7.14.1 Python properties ... 583
7.15 Profitina Laundry: the Service Hierarchy and Polymorphic Pricing ... 585
7.16 Profitina in Practice ... 597
7.17 Common Mistakes, and How to Avoid Them ... 598
7.18 Chapter Summary ... 599
7.19 Exercises ... 599
7.20 Solutions to the Exercises ... 601
References ... 613

CHAPTER 8 Evaluation and Discussion II: Recursive Structures, Robust Classes, and Class Hierarchies ... 615

8.1 Where We Left Off ... 615
8.2 How to Approach a Multi-Class Design Problem ... 616
8.3 The Two Structures of This Chapter: Trees and Cons Lists ... 618
8.3.1 A binary tree that does its work by delegation ... 618
8.3.2 An immutable list built from nil and cons ... 622
8.4 Problems on Binary Trees ... 630
8.4.1 Problem 8.1 — Building a tree by hand ... 630
8.4.2 Problem 8.2 — Tripling every node ... 635
8.4.3 Problem 8.3 — Printing the depth-first traversal ... 641
8.4.4 Problem 8.4 — A fresh copy of a tree ... 645
8.4.5 Problem 8.5 — Saving the traversal into an array ... 652
8.4.6 Problem 8.6 — Finding a value in a binary search tree ... 656
8.4.7 Problem 8.7 — Inserting a value into a binary search tree ... 661
8.5 Problems on Recursive Lists ... 668
8.5.1 Problem 8.8 — Negating a list ... 668
8.5.2 Problem 8.9 — Finding the position of an element ... 671
8.5.3 Problem 8.10 — Are all elements positive? ... 674
8.5.4 Problem 8.11 — Keeping only the positive elements ... 677
8.5.5 Problem 8.12 — Is the list sorted? ... 680
8.5.6 Problem 8.13 — Merging two sorted lists ... 683

8.5.7 Problem 8.14 — Skipping every copy of an element 686
8.5.8 Problem 8.15 — Removing duplicates 689
8.6 New Worked Problems: Exceptions, Files, and Arrays 692
8.6.1 Problem 8.16 — An exception-safe order file parser 692
8.6.2 Problem 8.17 — A documented array that grows itself 701
8.7 New Worked Problem: Interfaces and Their Tests 707
8.7.1 Problem 8.18 — One interface, two implementations, one set of tests 707
8.7.2 The same tests in JUnit 6 713
8.8 New Worked Problems: Inheritance, Polymorphism, and Access Control 714
8.8.1 Problem 8.19 — A class hierarchy traced by hand 714
8.8.2 Problem 8.20 — Refactoring two duplicated classes into a hierarchy 722
8.8.3 Problem 8.21 — An encapsulation bug hunt 730
8.9 The Problems of This Chapter Compared 736
8.10 Profitina in Practice 737
8.11 Common Mistakes, and How to Avoid Them 738
8.12 Chapter Summary 739
8.13 Exercises 739
8.14 Solutions to the Exercises 741
References 748

CHAPTER 9 Collections and Generics 749
9.1 Where We Left Off 749
9.2 Two Tools to Carry Along: Constants in Interfaces and Recursive Helpers 750
9.2.1 Constants in interfaces 750
9.2.2 Recursion, once more: base case and recursive call 754
9.2.3 Helper methods with extra parameters 756
9.3 The Collections Framework: a Map of the Territory 760
9.3.1 Three families: sets, lists, and maps 762
9.3.2 Program to the interface 762
9.3.3 Python's collections 762
9.4 Lists 763
9.4.1 The everyday operations 763
9.4.2 ArrayList versus LinkedList 767
9.4.3 Visiting every element, and removing safely 768
9.5 Sets 771
9.5.1 A set of numbers, and a set of names 771
9.5.2 How a HashSet finds an element: hashing 775
9.5.3 How a TreeSet keeps order 777
9.6 Maps 777
9.6.1 A map from keys to values 777
9.6.2 Reading a file into collections 782
9.7 Queues, Stacks, Deques, and Priority Queues 785
9.8 Generics from Zero 790
9.8.1 Life before generics 790
9.8.2 A generic class: GList 793
9.8.3 Generic methods, and a Pair at last 797
9.8.4 Bounded type parameters 802
9.8.5 Wildcards: why a List is not a List 809
9.8.6 Type erasure: what is left at run time 815
9.8.7 Generics in Java and Python: a summary 818
9.9 Putting Things in Order: Comparable and Comparator 819
9.9.1 Iterators, duplicates, iterables and a comparator 819
9.9.2 Comparators for real use 826
9.10 Choosing the Right Collection 831
9.11 Profitina Laundry: an Order Book with Several Indexes 833

9.12 Profitina in Practice 848
9.13 Common Mistakes, and How to Avoid Them 848
9.14 Chapter Summary 849
9.15 Exercises 850
9.16 Solutions to the Exercises 852
References 864

CHAPTER 10 Graphical User Interfaces and Event Handling 865
10.1 Where We Left Off 865
10.2 From the Console to the Window: Who Is in Charge? 866
10.3 Three Toolkits: AWT, Swing, and Tkinter 867
10.3.1 AWT: the original toolkit 867
10.3.2 Swing: components that draw themselves 868
10.3.3 Tkinter: Python's standard toolkit 869
10.4 Your First Window 870
10.4.1 Running a window without a screen: the self-test 871
10.5 Components, Containers, and Panels 873
10.5.1 The content pane 873
10.5.2 Panels, and why every window here is a class 874
10.5.3 The component tree of a real window 874
10.6 The Event Loop, Explained from Zero 875
10.6.1 Where events come from 875
10.6.2 The loop itself 875
10.6.3 An event loop you can read 876
10.7 Listening to Events 881
10.7.1 Reading a text field 881
10.7.2 The event object, and which component caused the event 885
10.7.3 Item events: check boxes 885
10.7.4 The four kinds of events in this chapter 889
10.8 Layout Managers versus pack and grid 889
10.8.1 FlowLayout: like words in a paragraph 890
10.8.2 BorderLayout: five regions 890
10.8.3 BoxLayout: a single row or column 890
10.8.4 GridBagLayout: the most flexible 891
10.8.5 Tkinter: pack, grid and place 892
10.8.6 Watching the managers work 892
10.9 Painting: paintComponent and the Canvas 900
10.9.1 The coordinate system, and the Graphics object 900
10.9.2 Color 901
10.9.3 Who calls paintComponent? 901
10.9.4 A box that darkens and lightens 902
10.9.5 The Tkinter Canvas: shapes that remember themselves 907
10.10 Inner Classes, Anonymous Classes, and Lambdas 909
10.10.1 Inner classes 909
10.10.2 Anonymous classes and lambdas 909
10.10.3 Five styles side by side 910
10.10.4 Effectively final: why Java copies 914
10.10.5 Python: bound methods, closures, lambdas, partial, callable objects 914
10.10.6 Drawing with the mouse 918
10.11 The Event Dispatch Thread and the Thread Rule 922
10.11.1 One thread owns the window 922
10.11.2 Slow work, the wrong way and the right way 923
10.12 Profitina Laundry: The Order-Entry Window 930
10.12.1 The window, described precisely 931
10.12.2 The prices, precisely 932

10.12.3 The design: a model that knows nothing about windows 933
10.12.4 The Swing program 934
10.12.5 The self-test 943
10.12.6 The Tkinter program 944
10.13 Profitina in Practice 953
10.14 Common Mistakes, and How to Avoid Them 954
10.15 Chapter Summary 955
10.16 Exercises 955
10.17 Solutions to the Exercises 957
References 967

CHAPTER 11 The Model–View–Controller Pattern 969

11.1 Where We Left Off 969
11.2 A Story: The Front Desk That Knew Too Much 970
11.3 Three Roles: Model, View, Controller 971
11.3.1 A model, a view and a controller you can point at 973
11.3.2 Who may know whom 974
11.3.3 Where the idea came from 975
11.4 A First MVC Program: The Student Record 976
11.5 The Observer Pattern Underneath MVC 981
11.5.1 Java: from `Observable` to `PropertyChangeSupport` 982
11.5.2 Python: an observer in a dozen lines 985
11.5.3 One change, many calls 988
11.5.4 Push or pull? 989
11.6 The Counter, Rebuilt: MVC with a Window 989
11.6.1 Reading the program 994
11.6.2 The window 994
11.6.3 The self-test 995
11.7 The Phone Keypad: Predictive Text in MVC 999
11.7.1 Words, keys and signatures 999
11.7.2 The program 1000
11.7.3 The window 1008
11.7.4 The self-test 1009
11.7.5 The same keypad in Python 1010
11.8 Testing the Model Without the GUI 1016
11.9 Profitina Laundry Front Desk: A Complete MVC Application 1022
11.9.1 The rules of the model 1022
11.9.2 The design 1023
11.9.3 The model 1024
11.9.4 The controller and the self-test 1031
11.9.5 The Swing views 1039
11.9.6 The same desk in Python 1044
11.10 Unit Tests for the Front Desk Model 1053
11.11 Variations: MVP and MVVM 1057
11.12 Profitina in Practice 1059
11.13 Common Mistakes, and How to Avoid Them 1059
11.14 Chapter Summary 1060
11.15 Exercises 1061
11.16 Solutions to the Exercises 1062
References 1073

CHAPTER 12 Evaluation and Discussion III: Predictive Text with Lists, Maps, Trees, and a Keypad 1074

12.1 Where We Left Off 1074
12.2 How to Approach a Collections-and-Interface Problem 1075

12.3 The Predictive-Text Problem ... 1076
12.3.1 Keys, letters and signatures ... 1076
12.3.2 The dictionary file ... 1077
12.3.3 Packages and how to run the programs ... 1077
12.3.4 The whole design at a glance ... 1078
12.4 Part One: Prototypes ... 1078
12.4.1 Problem 12.1 — From a word to its signature ... 1078
12.4.2 Problem 12.2 — From a signature to its words, the slow way ... 1088
12.4.3 Problem 12.3 — Two command-line programs ... 1089
12.5 Part Two: A Sorted List and an Interface ... 1092
12.5.1 Problem 12.4 — A sorted list of pairs and binary search ... 1092
12.5.2 Problem 12.5 — The `Dictionary` interface ... 1102
12.5.3 Problem 12.6 — The list program, and the first timing ... 1104
12.6 Part Three: A Map and a Tree ... 1113
12.6.1 Problem 12.7 — A multi-valued map ... 1113
12.6.2 Problem 12.8 — A tree of keys that also matches prefixes ... 1119
12.6.3 Problem 12.9 — One line to change, and the full race ... 1129
12.7 Part Four: A Keypad in MVC ... 1133
12.7.1 Problem 12.10 — The predictive-text GUI ... 1133
12.8 New Worked Problems: Collections and Generics ... 1149
12.8.1 Problem 12.11 — Choosing a collection by measuring it ... 1149
12.8.2 Problem 12.12 — A generic class with a bounded type parameter ... 1155
12.8.3 Problem 12.13 — A chain of comparators ... 1162
12.9 New Worked Problems: Events, Observers, and MVC ... 1167
12.9.1 Problem 12.14 — An event-driven counter with a tested model ... 1167
12.9.2 Problem 12.15 — Two observer bugs ... 1176
12.9.3 Problem 12.16 — Buttons made in a loop ... 1182
12.10 The Problems of This Chapter Compared ... 1187
12.11 Profitina in Practice ... 1188
12.12 Common Mistakes, and How to Avoid Them ... 1189
12.13 Chapter Summary ... 1190
12.14 Exercises ... 1190
12.15 Solutions to the Exercises ... 1191
References ... 1199

CHAPTER 13 Iterators, Immutability, and More Collections ... 1200
13.1 Where We Left Off ... 1200
13.2 Walking Through a Collection: Iterators ... 1201
13.2.1 The Iterator protocol in Java ... 1201
13.2.2 The same protocol in Python ... 1204
13.2.3 What an iterator cannot do ... 1206
13.3 Iterable, the Enhanced for Loop, and ListIterator ... 1207
13.3.1 Removing adjacent duplicates ... 1207
13.3.2 ListIterator: backwards, replace, insert, remove ... 1210
13.4 Writing Your Own Iterable Collection ... 1212
13.4.1 Java: Iterable plus an inner iterator class ... 1212
13.4.2 Python: iter, next, and generators ... 1215
13.4.3 An iterable is not an iterator ... 1218
13.5 Changing a Collection While Walking It ... 1218
13.6 Putting Objects in Order: Comparable and Comparator ... 1224
13.6.1 The contract every comparison must keep ... 1224
13.6.2 A comparator by age ... 1225
13.6.3 Names in three orders, and a comparison that only looks right ... 1229
13.7 Copies: Aliasing, Shallow and Deep ... 1234
13.8 Immutability from Zero ... 1239

13.8.1 Why would anyone want an object that cannot change? 1239
13.8.2 final is not immutable; views and copies 1243
13.8.3 Writing an immutable class 1247
13.9 Value Objects, Equality and Hashing 1253
13.10 Streams, Comprehensions, and itertools 1259
13.11 Profitina Laundry: Money, an Order History, and a Report Pipeline 1264
13.12 Profitina in Practice 1278
13.13 Common Mistakes, and How to Avoid Them 1279
13.14 Chapter Summary 1280
13.15 Exercises 1281
13.16 Solutions to the Exercises 1283
References 1292

CHAPTER 14 Threads, Races, Deadlock, and Livelock 1294

14.1 Where We Left Off 1294
14.2 What Is a Thread? 1295
14.2.1 Concurrency and parallelism 1295
14.2.2 Why use threads at all? 1296
14.3 A Parent Who Cannot Get Back Control 1297
14.3.1 The children as threads 1297
14.3.2 Stopping with a flag, and the visibility trap 1302
14.4 Four Ways to Create a Thread 1305
14.5 The Life of a Thread 1310
14.6 Python's Global Interpreter Lock, and When Threads Help 1314
14.6.1 The Global Interpreter Lock 1315
14.7 Race Conditions: When Two Threads Share a Counter 1318
14.7.1 Why `c++` is not one step 1321
14.7.2 The same race in Python 1322
14.8 Mutual Exclusion: `synchronized`, Locks, and Atomics 1324
14.8.1 Why a lock also fixes visibility 1329
14.8.2 Locks in Python 1329
14.9 Waiting for a Condition: `wait`, `notifyAll`, and `Condition` 1331
14.10 Producer and Consumer the Easy Way: Blocking Queues 1339
14.11 Deadlock 1341
14.11.1 The dining philosophers 1341
14.11.2 Two clerks and two machines 1348
14.12 Preventing Deadlock 1351
14.13 Livelock and Starvation 1357
14.13.1 Livelock 1357
14.13.2 Starvation and fairness 1363
14.14 Profitina Laundry: Machines as Worker Threads 1363
14.15 Profitina in Practice 1372
14.16 Common Mistakes, and How to Avoid Them 1373
14.17 Chapter Summary 1374
14.18 Exercises 1375
14.19 Solutions to the Exercises 1376
References 1382

CHAPTER 15 Sockets, Networking, and the Team Project 1384

15.1 Where We Left Off 1384
15.2 Networking from Zero 1385
15.2.1 Hosts and IP addresses 1385
15.2.2 Ports 1385
15.2.3 Layers: how a message travels 1386
15.2.4 TCP and UDP 1387

15.2.5 A first network program: datagrams to ourselves 1388
15.2.6 Clients and servers 1391
15.3 Sockets in Java and Python 1391
15.4 A Line-Based Echo Server and Client 1393
15.4.1 The server 1393
15.4.2 The client 1396
15.4.3 Running the pair in two terminals 1398
15.4.4 Lines, flushing and framing 1399
15.4.5 When the other side is not there 1399
15.5 Designing a Protocol on Paper First 1400
15.6 Serving Many Clients at Once 1406
15.6.1 A thread per client, from a pool 1406
15.6.2 The multi-client knock-knock server 1406
15.6.3 The client, and three terminals 1412
15.6.4 Proving that the server is concurrent 1415
15.7 Profitina Laundry: an Order-Status Service in JSON over TCP 1419
15.7.1 The specification, on paper 1419
15.7.2 JSON in both languages 1421
15.7.3 The server 1429
15.7.4 The client object 1444
15.7.5 Java talks to Python, and Python talks to Java 1451
15.7.6 The cost of a conversation 1455
15.8 HTTP: The Protocol of the Web 1455
15.8.1 What an HTTP conversation looks like 1456
15.8.2 An HTTP server and client in each language 1456
15.9 Security and Personal Data 1460
15.9.1 Seven habits for networked code 1461
15.9.2 What UU PDP asks of a program like ours 1461
15.10 Testing Networked Programs 1462
15.11 The Team Project 1469
15.11.1 Choosing a topic 1469
15.11.2 Splitting the work 1470
15.11.3 From specification to design to implementation 1470
15.11.4 Test cases across the interface 1471
15.11.5 Working together with Git 1471
15.11.6 The README 1472
15.11.7 Failure: the server goes down, a client goes down 1473
15.11.8 Documentation that people use 1473
15.11.9 Presenting the project 1474
15.11.10 A complete sample project brief 1474
15.11.11 A checklist for the team 1475
15.12 Profitina in Practice 1476
15.13 Common Mistakes, and How to Avoid Them 1476
15.14 Chapter Summary 1477
15.15 Exercises 1478
15.16 Solutions to the Exercises 1479
References 1497

CHAPTER 16 Evaluation and Discussion IV: Iterators, Concurrency, Networking, and the Whole Book 1499

16.1 Where We Left Off 1499
16.2 How to Approach a Whole-System Project 1501
16.3 The Project Brief, Restated 1502
16.4 Problems on Planning and Protocol Design 1503
16.4.1 Problem 16.1 — A topic, a division of work, and the first contract 1503

16.4.2 Problem 16.2 — The protocol as a flow chart and a state machine 1512
16.5 Problems on Building and Testing the Client–Server System 1521
16.5.1 Problem 16.3 — A server for many clients, and a console client 1521
16.5.2 Problem 16.4 — Each side tests the other, with stubs 1534
16.5.3 Problem 16.5 — Many clients at once: a multithreaded, distributed test 1540
16.6 Problems on Robustness, the Interface, and Documentation 1548
16.6.1 Problem 16.6 — When the server or a client goes down 1548
16.6.2 Problem 16.7 — A window that never freezes 1555
16.6.3 Problem 16.8 — Performance criteria, product documentation, and the report 1568
16.7 New Worked Problems: Iterators and Immutable Values 1573
16.7.1 Problem 16.9 — A pickup schedule with its own iterator 1573
16.7.2 Problem 16.10 — Money as an immutable value object 1582
16.8 New Worked Problems: Threads 1591
16.8.1 Problem 16.11 — A shared counter, a measured race, and its cure 1591
16.8.2 Problem 16.12 — A producer–consumer wash line 1598
16.9 New Worked Problem: A Tiny Protocol of Our Own 1608
16.9.1 Problem 16.13 — LAUNDRY/1, a request–reply protocol 1608
16.10 The Capstone: Profitina Laundry from End to End 1619
16.10.1 Problem 16.14 — A small but complete laundry system 1619
16.11 The Problems of This Chapter Compared 1644
16.12 The Whole Book in One Table 1645
16.13 Where to Go Next 1646
16.13.1 Design patterns 1646
16.13.2 Build tools and environments 1647
16.13.3 Frameworks 1648
16.14 Profitina in Practice 1648
16.15 Common Mistakes, and How to Avoid Them 1649
16.16 Chapter Summary and a Closing Word 1650
16.17 Exercises 1650
16.18 Solutions to the Exercises 1651
References 1659

List of Figures

Figure 1.1 How source code becomes a running program: Java compiles once to bytecode that the Java Virtual Machine runs; Python compiles to bytecode each time it runs, inside the interpreter. 4
Figure 1.2 The class `Order` (top) is the blank form: it lists the state every order holds and the behavior every order offers; the objects `ani` and `budi` (bottom) are two filled-in tickets. 6
Figure 1.3 One morning at the counter as a conversation between objects: solid arrows are messages, dashed arrows are answers, and the numbers give the order in which they happen. 7
Figure 1.4 The same three orders organized two ways: on the left, four parallel arrays that any line of the program can write to; on the right, one object per order, whose state only its own methods can change. 11
Figure 1.5 The most used languages in the Stack Overflow Developer Survey 2025, all respondents: Python is fourth and Java seventh of the fifteen shown. 13
Figure 1.6 The PYPL ranking for April 2026, based on how often each language's tutorials are searched for: Python first, Java third. 14
Figure 1.7 The TIOBE ranking for September 2026, based on engineers, courses and vendors per language: Python first, Java fourth. 14
Figure 1.8 The GitHub Octoverse 2025 ranking by monthly contributors: TypeScript, Python and JavaScript lead, and Java is fourth. 15
Figure 1.9 The `bin` folder of a JDK, listed in a Linux terminal; the five tools of **Table 1.3** are highlighted, and on Windows each file ends in `.exe`. 17
Figure 1.10 Opening the Command Prompt on Windows: after typing `cmd` in the taskbar search box, choose the best match (1). 17
Figure 1.11 Adding the JDK's `bin` folder to the front of the PATH for the current Command Prompt window with the `set path=` command. 18
Figure 1.12 The life of an order: each message moves it one step forward, and a message sent at the wrong moment, such as `pickUp()` while the laundry is still washing, is refused and changes nothing. 30
Figure 2.1 From source code to the processor. Java compiles once with javac and the JVM runs the bytecode every time; Python compiles inside the interpreter every time the program starts, then its virtual machine runs the bytecode. 53
Figure 2.2 The Eclipse Installer's list of packages. Eclipse IDE for Java Developers is the one to choose. 57
Figure 2.3 Choosing the Java runtime for Eclipse and the installation folder, just before pressing INSTALL. 58
Figure 2.4 The Eclipse IDE Launcher asks for a workspace folder before the workbench opens. 58
Figure 2.5 File > Switch Workspace > Other… opens the launcher again so that you can choose a different workspace. 59
Figure 2.6 The Appearance preferences page, where the color theme is chosen. 60
Figure 2.7 Window > Perspective > Open Perspective, with the Debug and Java Browsing perspectives offered directly and the rest under Other… 61
Figure 2.8 The Open Perspective dialog lists every perspective of the installation; Java is marked as the default. 61
Figure 2.9 The Java perspective of an empty workspace: Package Explorer on the left, editor area in the middle, Outline on the right, Problems, Javadoc and Declaration along the bottom. 62
Figure 2.10 The Show View dialog, grouped by category; type part of a view's name in the filter box to find it. 63
Figure 2.11 The Problems view with two warnings; the context menu offers Quick Fix. 64
Figure 2.12 The Javadoc view showing the documentation of println(String) while the cursor is on a call to it. 64
Figure 2.13 A right-click in the editor's left ruler: Show Line Numbers is ticked, and Toggle Breakpoint sits at the top. 65

Figure 2.14 The New Java Project wizard, with separate folders for sources and class files. 66
Figure 2.15 Right-click on src, then New > Package. 67
Figure 2.16 The New Java Package dialog: the source folder and the package name. 67
Figure 2.17 The New Java Class dialog, with the main method stub ticked. 68
Figure 2.18 Run As > Java Application from the context menu of the class. 69
Figure 2.19 The Console view after the run. The first line records which program ran, on which Java, and when. 69
Figure 2.20 The Export wizard: JAR file under the Java category. 70
Figure 2.21 The JAR File Specification page: which resources to export and where to write the JAR. 71
Figure 2.22 Running the class from the JAR in a Windows Command Prompt. 71
Figure 2.23 A map of the VS Code window while debugging: Explorer and Java Projects in the side bar, the editor with a breakpoint in the gutter, the Run and Debug views, the debug toolbar, the panel, and the interpreter selector in the status bar. 72
Figure 2.24 The starting version of Counter.java: TODO comments describe what count(x) must do, and it returns 0 until they are done. 77
Figure 2.25 Counter cannot be resolved to a type: the quick-fix box offers to import util.Counter, or the unrelated org.w3c.dom.css.Counter. 79
Figure 2.26 Organize Imports asks which Counter is meant when the short name is ambiguous. 80
Figure 2.27 Exporting two projects to an archive file, keeping the directory structure. 81
Figure 2.28 Importing existing projects from an archive file into the workspace. 82
Figure 2.29 The Package Explorer's view menu, with Link with Editor about to be switched on. 82
Figure 2.30 Toggle Breakpoint in the context menu of the editor's left ruler (Ctrl+Shift+B). 85
Figure 2.31 A breakpoint on line 8, Counter counter = new Counter(); the dot in the ruler marks it. 85
Figure 2.32 Debug As > Java Application starts the program under the debugger. 86
Figure 2.33 The Debug toolbar button reruns the most recent debug launch, here Tester. 86
Figure 2.34 Eclipse offers to open the Debug perspective when the program first suspends. 87
Figure 2.35 The execution buttons of the Eclipse debugger with their keys: Resume (F8), Terminate (Ctrl+F2), Step Into (F5), Step Over (F6) and Step Return (F7). 87
Figure 2.36 Stepping seen on the call stack. Step Into goes down into the called method, Step Out returns to the caller, and Step Over runs the whole call and stays at the same level. 88
Figure 2.37 The Debug view shows the thread main suspended in Tester.main at line 11; each line under the thread is one frame of the call stack. 88
Figure 2.38 The Variables view after count(5) returned: args, counter and result = 15. 89
Figure 2.39 Changing a value during debugging: the loop variable i in count(int) is being edited in the Variables view. 89
Figure 2.40 Without a toString() method, the detail pane shows only util.Counter@71bbf57e. 90
Figure 2.41 A Detail Formatter for util.Counter that displays getResult() instead. 90
Figure 2.42 The Breakpoints view lists every breakpoint; the check boxes enable or disable them, and the lower part holds the Hit count and Conditional properties. 91
Figure 2.43 Skip All Breakpoints (Ctrl+Alt+B) disables every breakpoint without deleting any. 91
Figure 2.44 The debugging cycle. When the comparison in step 5 refutes the hypothesis, go back to step 3 with what you have learned, not back to guessing. 105
Figure 3.1 The class `Hello` open in the Eclipse editor: one class, one `main` method, one statement. 121
Figure 3.2 `Hello2` in Eclipse: the program reads the first command-line word from `args[0]` if one was given 122
Figure 3.3 The class `Dog` as a blueprint (left) and two objects built from it with `new` (right); each object has its own copy of every field. 126
Figure 3.4 The first part of `Dog.java` in Eclipse: the fields, the constructor and the first getters and setters. 128
Figure 3.5 The rest of `Dog.java` in Eclipse: the remaining getters and setters and `toString`. 128
Figure 3.6 A JUnit test class `DogTest` in Eclipse; the method marked `@Test` creates a dog and uses `assertEquals` to compare the expected age with the actual one. 135
Figure 3.7 The Eclipse Package Explorer: `Dog.java` and `DogTest.java` in the project `dog`, with the JUnit library file under *Referenced Libraries*. 135

Figure 3.8 A red bar: `test1` fails with expected 3 but was 0, because the constructor of this `Dog` says `age = age;` instead of `this.age = age;`. 136
Figure 3.9 A green bar: after the fix, the test runs and passes. 136
Figure 3.10 Memory inside `main` of **Listing 3.13**: `a` and `b` hold values, while `fido`, `alias` and `twin` hold arrows; `fido` and `alias` point to the same object. 138
Figure 3.11 Names and objects in **Listing 3.14**: every name is an arrow; `fido` and `alias` are bound to the same `Dog` object. 140
Figure 3.12 String methods never change a string: `toUpperCase` and `+` build new strings, and assignment only moves the arrow. 145
Figure 3.13 Two different `String` objects with the same characters: `==` says `false`, `equals` says `true` 146
Figure 3.14 The error after adding 0.1 again and again, for `float` and `double`; the dips to the bottom are the moments when the rounding errors happen to cancel. 158
Figure 3.15 An `else if` chain as a flowchart: the conditions are tried from the top, and the first true one decides 164
Figure 3.16 Without `break`, execution falls through from the matching case into the next; with `break`, it leaves the `switch`. 166
Figure 3.17 The four parts of `for (int j = 0; j < 4; j++)`: initialization once, then condition, body and update in a cycle until the condition is false. 169
Figure 3.18 The balance of GBP 100 at 2.3% per year over 500 years, on a logarithmic scale; the straight line is the signature of compound growth. 182
Figure 3.19 The first two classes of Profitina Laundry and the program that uses them. 184
Figure 3.20 The price calculation of `Service.price`: chargeable weight, base price, then the two optional adjustments. 185
Figure 4.1 The eight steps used for every problem in this chapter; the dashed arrow is the loop between writing code and running the tests. 207
Figure 4.2 The clock at 3:00 (left) and at 9:00 (right); the amber arc runs counterclockwise from the hour hand (short, dark) to the minute hand (long, teal), giving 90 ∘ and 270 ∘. 216
Figure 4.3 The clock at 4:41: the hour hand has crept to 140.5 ∘, the minute hand points at 246 ∘, and the counterclockwise angle between them is 254.5 ∘, which rounds half to even to 254. 217
Figure 4.4 Lower box: the correct `getSum` leaves `f1` and `f2` alone and returns a new object. Upper box: the tempting version of Section 4.8.7 overwrites `f2` and returns it, so `f2` and the result are the same changed object. 240
Figure 4.5 The class diagram of Problem 4.10: a `LaundryReceipt` has a `Rupiah` total; the filled diamond marks that the total belongs to the receipt. 267
Figure 5.1 An array of five strings in memory. The variable holds a reference to the array object; the slots are numbered from 0 to 4, and index 5 does not exist. 305
Figure 5.2 What `add` does when the hidden array is full: a larger array is made, the old references are copied into it, and the old array is thrown away. 318
Figure 5.3 Propagation. The calls go down from `main`; the exception travels back up, abandoning each method on the way, until it reaches a matching `catch` 327
Figure 5.4 Part of Java's exception hierarchy. Blue classes are checked exceptions, yellow ones are unchecked, and red ones are errors that programs should not try to handle. 328
Figure 5.5 Part of Python's exception hierarchy. Programs catch subclasses of `Exception`; the exceptions beside it ask the program to stop. 329
Figure 5.6 The paths through a `try-catch-finally` statement. Every path passes through a `finally` block. 334
Figure 5.7 The order-loading pipeline. A bad line becomes a message in the error list and loading continues; a missing file or a wrong header stops the load and is reported by the application. 356
Figure 6.1 From documentation comments to documentation: `javadoc` builds linked HTML pages, `pydoc` prints help text or writes an HTML page, and `doctest` runs the examples written in docstrings. 395
Figure 6.2 Arrange, act, assert: build the object, call the one method under test, compare the expected value with the actual one. 403

Figure 6.3 The test-first cycle: a failing test (red), the simplest code that passes (green), then tidying with the tests as a safety net (refactor). 410
Figure 6.4 Two references to two different objects with the same contents: not identical, but equal if the class says so — and then their hash codes must be equal too. 411
Figure 6.5 The client depends only on the interface `DBops`; `LinearDB` and `TreeDB` implement it and can be exchanged without changing the client. 443
Figure 6.6 Moving up the type hierarchy (from `Name` to `Comparable` or `Object`) is automatic; moving down needs an explicit cast, which is checked when the program runs. 451
Figure 6.7 The pricing design: five classes implement `PriceRule`, two of them by wrapping another `PriceRule`; the demo program depends only on the interface. 469
Figure 7.1 The duplication problem and its cure. Left: two independent classes each carry their own copy of `addSeveral` and `findOneOf`. Right: the shared methods move up into the abstract superclass `DB`; each subclass keeps only what is genuinely its own. 508
Figure 7.2 Constructor chaining for `new Shirt("blue", true)`. The calls travel up the hierarchy to `Object` first (solid arrows); the constructor bodies then finish in reverse order (dashed arrows), so the most general part of the object is always ready before the most specific part. 525
Figure 7.3 UML class diagram of the Animal program. `Animal` has four overloaded constructors and four overloaded `getPrice` methods; `Cat`, `Cow` and `Dog` each override `sound()`, and `Cat` adds a method of its own. A hollow triangle points from a subclass to its superclass; `-` marks a private member and `+` a public one. 529
Figure 7.4 Memory while **Listing 7.9** runs. Local variables on the stack hold references to objects on the heap; every object's header points to its class; every class has one method table (*vtable*) whose slot *k* holds the code for the *k*-th overridable method. Subclasses copy their parent's slots and overwrite the ones they override. 546
Figure 7.5 Two hierarchies with abstract inner levels (names in italics). Left: geometric figures, split into open and closed figures. Right: students, split into taught and research students. 553
Figure 7.6 Part of the Java collections hierarchy. Solid lines with hollow triangles are `extends`; dashed lines are `implements`. The abstract classes supply shared code, the interfaces state the contracts, and `ArrayList` and `LinkedList` are the concrete classes programs actually create. 560
Figure 7.7 Inheriting from a concrete class. `TreeDB` receives `LinearDB`'s array and counter although a tree never uses them, and every `TreeDB` now claims to be a `LinearDB`. 563
Figure 7.8 The fragile base class. Left: the subclass's `addAll` calls the superclass's `addAll`, which calls `add`, which dynamic dispatch sends back to the subclass: every tag is counted twice. Right: the composed class calls methods of an object it merely holds; its own `add` is never re-entered. 568
Figure 7.9 The diamond. `WashAndIron` has two paths up to `Service`. Python linearizes the hierarchy into one method resolution order, and every `super()` call moves to the next class in that order (numbered arrows), so `Service.steps` runs exactly once. 573
Figure 7.10 Four classes in two packages. `Beta` shares a package with `Alpha`; `AlphaSub` is in another package but extends `Alpha`; `Gamma` has no relationship with `Alpha` at all. 576
Figure 7.11 UML class diagram of the Profitina Laundry price engine. `Service` and `WeightService` are abstract; five concrete services extend them; `ExpressService` both *is* a `Service` and *wraps* one (hollow diamond); an `Order` holds any number of services (filled diamond). 586
Figure 8.1 The binary tree of Problem 8.1. Node 4 has only a right child (8) and node 7 has only a left child (9); every other inner node has two children. 618
Figure 8.2 The objects behind `List a = cons(2, cons(-5, cons(8, nil())))`. Each cell has a head and a tail reference; the last tail is the single shared empty list. 622
Figure 8.3 UML class diagram of the list classes. `ListException` is unchecked because it extends `RuntimeException`; every problem class of Section 8.5 only uses `List`, it does not extend it. 623
Figure 8.4 The sample tree before and after `t.triple()`: the shape is the same and every value is multiplied by three. 636
Figure 8.5 Left: the original `t`. Right: the fresh copy `u`, seven new objects. Middle: the wrong copy `w`, one new object whose references point back into `t`. 646
Figure 8.6 The binary search tree used in Problems 8.6 and 8.7. The thick path is followed by `find(8)` and `find(10)`; the dashed leaves 9 and 12 are added by `insert(9)` and `insert(12)`. 657

Figure 8.7 UML class diagram of Problem 8.18. The dashed lines with hollow triangles mean "implements"; `quote` is a default method of the interface. 707
Figure 8.8 The service hierarchy of Problem 8.19. `Express` extends `Wash`, overrides `pricePerKg` and `describe`, and uses `super` to reach the `Wash` versions. 715
Figure 8.9 Before (left): two copies of the same fields and methods, each class using the other. After (right): one abstract `Fighter`, and subclasses that add only what differs. 723
Figure 8.10 Aliasing. Top: `LeakyOrder` and the caller share one list, so either can change it. Bottom: `SafeOrder` copied the list, so the caller's later changes do not reach the order. 730
Figure 9.1 The calls made by the standard recursive `fibonacci(5)`: fifteen calls, because the same subproblems are solved again and again — `fib(3)` twice and `fib(2)` three times. 755
Figure 9.2 The core of Java's Collections Framework. Interfaces are marked «interface»; dashed arrows mean *implements* and solid arrows mean *extends*; the grey classes are historical. `Map` is not a `Collection`; `Arrays` and `Collections` hold static helper methods. 761
Figure 9.3 Python's collection types and the abstract base classes of `collections.abc`, the counterpart of Java's collection interfaces. 763
Figure 9.4 The same list stored two ways. An `ArrayList` keeps references in one block of consecutive slots, with spare capacity at the end; a `LinkedList` keeps a chain of nodes, each linked to its neighbors. 767
Figure 9.5 A `HashSet` of four names. Each name's hash code selects one of 16 buckets, so `contains` looks in one bucket only; iteration walks the buckets in index order, which explains the printed order `[Sari, Budi, Agus, Rina]`. 776
Figure 9.6 The `TreeSet` of the same four names: a red-black search tree. An in-order walk visits the names in sorted order; a search follows one path from the root. 777
Figure 9.7 Three disciplines for taking elements out. A queue releases the oldest element (FIFO), a stack the newest (LIFO), and a priority queue the smallest, which a binary heap keeps at index 0 of its array. 786
Figure 9.8 Subtyping among classes (left) and among list types (right). `List<Cat>` is not a subtype of `List<Animal>`, but both are subtypes of `List<? extends Animal>`, and `List<Cat>`, `List<Animal>` and `List<Object>` are all subtypes of `List<? super Cat>`. 810
Figure 9.9 Comparisons made by one `contains` as n grows (both axes logarithmic): about $n/2$ for a list, about $\log_2 n$ for a balanced tree, and a small constant for a hash table with load factor $\alpha \leq 0.75$. 832
Figure 9.10 Choosing a collection with four questions. Each leaf names the Java class and the Python type that answer them. 833
Figure 9.11 The order book: four structures, one set of `LaundryOrder` objects. The class keeps an invariant: every order appears in `byId`, in its customer's list, and in exactly one status set. 834
Figure 10.1 The layers under a window. In Java, your panel is built from Swing components, which rest on AWT, which asks the operating system for a native window. In Python, `tkinter` is a thin layer of Python classes over the Tcl/Tk toolkit. 867
Figure 10.2 Part of the class hierarchies of Swing/AWT (left) and Tkinter (right). Shaded classes are the ones this chapter uses. Arrows point from a subclass to its superclass. 868
Figure 10.3 The component tree of the Profitina Laundry order-entry window. Each container arranges only its own children, with its own layout manager. 875
Figure 10.4 The life of an event. The operating system turns the user's actions into events and queues them; the event loop takes them one at a time, calls the listeners registered for them, and repaints when idle. A listener that changes what should be shown only *requests* a repaint. 876
Figure 10.5 A button and its listeners. The button keeps a list; registration appends to it; a click calls each listener's `actionPerformed` in turn. The button never knows what its listeners do. 881
Figure 10.6 The echo window after the user typed a greeting into the text field and pressed Enter: the listener has copied the text into the label. 883
Figure 10.7 The check-box window after *Mango* was ticked: the item listener rewrote the explanation label. 887
Figure 10.8 Five buttons in a `FlowLayout`: each keeps its preferred size, and the row flows from left to right. 890
Figure 10.9 The five regions of a `BorderLayout`. The center takes all the space the four edges leave over. 890

Figure 10.10 Two vertical boxes inside a horizontal box. The radio buttons on the left belong to one `ButtonGroup`, so only one can be chosen; the check boxes on the right are independent. 891
Figure 10.11 A `GridBagLayout` in which buttons span one cell, the rest of a row, or two rows, as set by `gridwidth`, `gridheight` and the weights. 891
Figure 10.12 Swing's coordinate system: the origin is the top-left corner and *y* grows downward. An oval is specified by its bounding box, not by its center. 901
Figure 10.13 The shading window after enough clicks on *Darken*: the box is black, and the *Darken* button is disabled because the level cannot go lower. 906
Figure 10.14 A variant that darkens with a button and lightens with a mouse click, and says so in a label; here the box has reached black and the label reports that it cannot be darkened further. 906
Figure 10.15 What listeners remember. Java inner and anonymous objects keep a hidden reference to their outer object; a lambda object holds copies of the values it captured. A Python closure keeps *cells* that refer to the variables of the function that created it. (The field names are the compilers' implementation details.) 914
Figure 10.16 A heart drawn by dragging the mouse, with the coordinates of the pointer shown in the label. 920
Figure 10.17 The thread rule on a time line. Top: the tally runs on the EDT, which cannot handle clicks or repaints for 600 ms. Bottom: a worker thread runs the tally while the EDT keeps handling events; `done()` returns to the EDT to show the result. 927
Figure 10.18 The Swing order-entry window with three orders, a fourth being typed (the live price already shows Rp68,250), and the second row selected. 931
Figure 10.19 The Tkinter order-entry window in the same state. (The screenshot shows the window's contents without the title bar that the operating system adds.) 932
Figure 10.20 The design of the order-entry program. The view owns an `OrderBook` and uses `OrderForm` for every calculation and check; the model classes never mention a window. 933
Figure 10.21 The oval program after a click: a small circle is drawn around the point that was clicked 956
Figure 11.1 The same front desk before and after MVC. Left: one window class does six jobs. Right: the model holds the orders and the rules, the view draws, and the controller translates gestures into model calls; the dashed arrow is the Observer notification by which the model tells the view that something changed. 971
Figure 11.2 The classic MVC diagram. The model encapsulates the application state, answers state queries, exposes the application's functionality and notifies views of changes; the view renders the model and sends user gestures to the controller; the controller maps user actions to model updates and selects the view for the response. 972
Figure 11.3 Model, view and controller of a small document editor. The model is five plain values; the view is a properties window that displays them; the controller offers one method per property, which the view calls when the user edits a field. 973
Figure 11.4 One model, two views. The *Display* window draws the text as it will appear on the page, and the *Properties* window shows the same values as numbers; both are kept current by the model's change notifications. 974
Figure 11.5 The controller as a mediator. User actions travel from the view to the controller, which updates the model's properties; the model's property changes travel back to the controller, which updates the view's components. The model and the view never refer to each other. 975
Figure 11.6 Class diagram of the student example. `StudentController` holds a `Student` model and a `StudentView`; it updates the model through setters and updates the view with `updateView()`; the demo class `StudentMvcDemo` uses the controller and retrieves the first record from a (simulated) database. 976
Figure 11.7 The Observer pattern as used by `ObserverDemo.java`. The subject `WashCounter` delegates its list of listeners to a `PropertyChangeSupport`, which notifies any number of objects implementing `PropertyChangeListener`; the two observers are lambdas. In Python the same roles are played by a plain list of callables. 982
Figure 11.8 Sequence diagram of one `increment()`. The counter calls its own `setLoads(1)`, which asks the support object to fire one event; the support object calls each listener in registration order, and only then does `increment()` return to `main`. 988

Figure 11.9 The Up/Down counter at work. Top row, started at 0: after clicks on *Up* the field shows 1, 2 and 10; after clicks on *Down* it shows 5 and −15. Bottom row, started at 100: 101, then 74. The mouse pointer shows which button was clicked last. 989

Figure 11.10 The classes of the predictive keypad. The keypad panel reports each press to the controller, which calls one of the model's four operations; the model uses the dictionary and notifies its three views, which read the text and the current match back from the model. 1000

Figure 11.11 The life of a laundry order. Each `advance()` moves an order one state to the right; an order that has been picked up refuses to advance. 1023

Figure 11.12 Class diagram of the front desk. The model (left) imports nothing from the GUI. The controller implements `OrderBookListener`, calls the model and updates three view contracts; the real Swing or Tkinter views and the text views of the self-test both implement those contracts. 1023

Figure 11.13 Sequence diagram of a click on *Add order*. The form reports the click to the controller, which reads the form, parses the weight and calls the model; the model validates, stores and notifies the controller, which refreshes the list and the summary; after the model returns, the controller clears the form and shows a message. 1039

Figure 11.14 MVC and two of its descendants. In MVC the view observes the model and sends gestures to the controller. In MVP a passive view reports events to a presenter, which sets every field of the view. In MVVM the view is bound, in both directions, to a view model that holds the screen's state and commands. 1058

Figure 12.1 The telephone keypad. Eight keys carry letters; keys 7 and 9 (shaded) carry four letters each, the other six carry three; key 0 is the space. 1076

Figure 12.2 UML class diagram of the predictive-text project. Dashed lines with hollow triangles mean "implements"; solid arrows mean "has" or "uses"; `DictionaryTreeImpl` refers to objects of its own class. 1078

Figure 12.3 The tree dictionary for eight words. Each word is stored in every node on the path of its signature, so the node `46` holds all four words that begin with the keys 4 and 6. 1120

Figure 12.4 Microseconds per lookup against dictionary size, on logarithmic axes, measured with `TimingComparison` (left) and `timing_comparison.py` (right). The prototype rises with the size; the map stays flat; the list and the tree lie in between. 1132

Figure 12.5 The Model–View–Controller pattern. The controller mediates: it turns user actions into updates of the model, and the model's changes reach the view. 1134

Figure 12.6 The window of an earlier version of this program, rebuilt with its hand-typed key labels, before any key is pressed. The label of key 9 is wrong: it reads `vwxy` instead of `wxyz`. 1134

Figure 12.7 The Swing window of **Listing 12.38** at the start (left), after pressing 4, 6, 6, 3 (middle), and after `* * * 0 9 6 7 5 3 #` (right). Key 1 is disabled because it carries no letters. 1142

Figure 12.8 The Tkinter window of **Listing 12.43** at the same three moments as in **Figure 12.7**. 1148

Figure 13.1 An iterator's cursor sits between elements. Each `next()` jumps over one element and returns it (grey boxes are already returned); after the last jump, `hasNext()` is `false`. 1203

Figure 13.2 Why Java throws `ConcurrentModificationException`. After `remove("a")` the elements shift left, but the iterator's cursor does not; there is no right answer to "what comes next?", so the iterator refuses to answer. 1219

Figure 13.3 Three kinds of "copy" of a list of persons. Aliasing: two variables, one list. Shallow copy: two lists, but their slots refer to the same `Person` objects, so renaming through one list is visible through the other. Deep copy: two lists and separate `Person` objects. 1235

Figure 13.4 A mutated key is lost. The entry was filed in bucket 994 mod 16 = 2; the changed key's hash sends lookups to bucket 1056 mod 16 = 0, which is empty, while a lookup with the old value finds the right bucket but fails the `equals` test. 1256

Figure 13.5 A filter–map–reduce pipeline. The source loads flow through a filter and a map into a reduction that folds them into one number; the same pipeline in one line of Java and one line of Python. 1259

Figure 13.6 UML class diagram of the laundry ledger. `Money`, `Price`, `OrderLine` and `Order` are records (immutable values); `OrderHistory` holds an unmodifiable list of orders, implements `Iterable<Order>`, and hands out two kinds of iterator. 1265

Figure 14.1 Profitina Laundry as a concurrent system. The front desk puts orders into one shared queue; three washing machines, each a worker thread, take orders from it; all three pay into one ledger, which accepts one machine at a time. 1295
Figure 14.2 Concurrency versus parallelism. Top: one core; the scheduler gives each thread a short time slice in turn, so all three make progress but only one runs at any instant. Bottom: two cores; two threads really run at once, and the scheduler moves threads between cores as it likes. 1296
Figure 14.3 The classic life cycle of a Java thread. A thread is born, becomes ready when started, runs when the scheduler dispatches it, and can leave the running state to wait, sleep or block on input and output before it becomes ready again or dies. 1310
Figure 14.4 The six values of `Thread.State` and the events that move a thread between them. `RUNNABLE` covers both "ready" and "running" of **Figure 14.3**; `BLOCKED` means waiting for a lock, `WAITING` means waiting without a time limit, and `TIMED_WAITING` means waiting with one. 1310
Figure 14.5 The GIL in pictures. Top: two computing threads take turns holding the GIL, so they never compute at the same moment. Bottom: threads that mostly wait release the GIL while waiting, so their waits overlap. 1317
Figure 14.6 A lost update, step by step. Child 1 reads 0; Child 2 reads 0 and writes 1; Child 1 writes its own 1 over it. Two increments have happened and the counter says 1. Two more steps later a decrement is lost as well, and the counter ends at -1 instead of 0. 1321
Figure 14.7 The five-cell circular buffer when it is full. Both indexes point to cell 2: the consumer's next value is the 3 stored there, and the producer's next value will go there as soon as the 3 has been taken. 1332
Figure 14.8 The output of `ProCon.java` shown in a Swing text area: the producer writes 1 to 10 into five circular cells, waits when the buffer is full, and the consumer's total is 55. 1336
Figure 14.9 The dining philosophers: five philosophers, five plates of spaghetti, and one fork between each pair of neighbors. Philosopher Pi needs forks Fi and $F(i + 1) \bmod 5$, as in `Philosopher.java`. 1342
Figure 14.10 Timeline of the deadlock of **Listing 14.21**. At $t1$ Ani takes the washer, at $t2$ Budi takes the dryer, at $t3$ and $t4$ each asks for the other's machine and blocks forever. The wait-for graph on the right has a cycle, and a cycle in the wait-for graph is exactly a deadlock. 1349
Figure 14.11 Livelock and its cure. Top: two always-polite workers hand the resource back and forth and never work. Bottom: each worker gives way at most once, so after two handovers both do their work. 1360
Figure 14.12 Class diagram of the concurrent laundry. The front desk (`main`) puts `Order` records into a `BlockingQueue`; three `WashingMachine` objects, each run by its own thread, take orders from it and record their prices in the single, fully synchronized `Ledger`. 1364
Figure 15.1 One request travelling through the four Internet layers. The client's layers wrap the JSON line in a TCP segment, an IP packet and a link-layer frame; the server's layers unwrap it in reverse. Each layer appears to talk directly to its twin on the other host. 1387
Figure 15.2 A client makes a connection request to the port on which the server is listening (here port 5050; the client's own port, such as 52100, is an ephemeral port chosen by its system). 1391
Figure 15.3 After acceptance the client and the server are connected through a new socket on the server's port 5050; the server's listening socket is still open for other clients. 1391
Figure 15.4 The life of one TCP connection. The server creates a listening socket and blocks in `accept()`; the client's connect triggers the three-way handshake; `accept()` returns a new connected socket; lines flow both ways; the client's `close()` sends FIN, which the server sees as the end of its input stream. 1393
Figure 15.5 The knock-knock protocol as a state machine. Each arrow is labeled "what the client said / what the server answers"; grey arrows handle unexpected input; the green arrow starts the next joke. 1401
Figure 15.6 A multi-client server. The acceptor thread only accepts connections and hands each new socket to a worker of a fixed-size thread pool; each worker owns one socket and one protocol object; the only state the workers share is guarded by an atomic counter or a lock. 1406
Figure 15.7 A conversation with the order-status service. Requests travel as JSON lines; the server consults the shared `OrderBook`, which is guarded by a lock; a malformed line receives an error,

not silence; `bye` ends the conversation. The client may be Java and the server Python, or the other way round. 1420
Figure 15.8 The five components of a team project, their owners, and the three interface documents between them. Dashed arrows show cross-testing: each owner writes tests for the component that uses theirs, or that theirs uses. 1470
Figure 15.9 Feature branches. The team starts `main` with the README and the protocol document; each member works on a branch; every branch returns to `main` through a reviewed pull request; the demonstrated version is tagged. 1472
Figure 16.1 The five parts of the Knock-Knock Club and the contracts between them. Solid arrows are "uses"; dashed arrows show that the testing member tests every contract. 1504
Figure 16.2 The knock-knock protocol as a flow chart: the conversation for people, with only the expected answers. 1513
Figure 16.3 The same protocol as a state machine. Every state has an arrow for every possible input, including "anything else". 1513
Figure 16.4 The threads of the knock-knock server. Each client has its own worker thread and its own protocol object; only immutable or thread-safe objects are shared. 1521
Figure 16.5 Cross testing through an interface. The client member writes tests for the server code; the server member writes tests for the client code. Each uses a stub of its own side. 1535
Figure 16.6 The Knock-Knock Club window (Swing) during a conversation. The status line reads "Connected"; the input field is enabled because the server is waiting for an answer. 1556
Figure 16.7 A lost update. Both machines read 41 before either writes, so both write 42: two loads were washed, and the counter rose by one. 1591
Figure 16.8 The wash line: one producer, a bounded buffer of capacity 2, and two consumers. Two `STOP` pills, one per machine, end the day. 1599
Figure 16.9 UML class diagram of the capstone. Hollow triangles mean "extends" (solid line) or "implements" (dashed line); the diamond means that the shop holds the orders. 1620

List of Tables

Table 1.1 The things in a laundry shop, each described by what it knows and what it can do. 5
Table 1.2 Java and Python side by side: the differences a beginner meets first. 16
Table 1.3 The tools of the Java Development Kit that this book uses. 16
Table 1.4 Four development environments for Java and Python. 19
Table 1.5 The workbench on three operating systems, from an empty computer to running programs in both languages. On Fedora use `sudo dnf install java-25-openjdk-devel python3.14`; on Arch, `sudo pacman -S jdk-openjdk python` 20
Table 1.6 Common problems when setting up Java, Python and VS Code, and how to fix them. 28
Table 1.7 The sixteen chapters of this book and what each one adds. 37
Table 2.1 The five jobs every IDE does, and where this chapter covers each. 51
Table 2.2 The same ideas under two names: a dictionary between Eclipse and VS Code. 52
Table 2.3 Keyboard shortcuts worth learning in the first week, in Eclipse and in VS Code. 83
Table 2.4 The debugger's execution controls and their keys and commands in Eclipse, VS Code, jdb and pdb. 87
Table 2.5 Bu Sari's bill worked out by hand. These are the values the program should reach, item by item. 95
Table 3.1 A class seen as a spreadsheet: the header row is the class `Dog`, each row below it is one `Dog` object. 125
Table 3.2 The parts of a class in Java and in Python. 129
Table 3.3 Identity, equality, and "no object" in the two languages. 140
Table 3.4 Everyday string operations in Java and Python. 144
Table 3.5 The format placeholders used in this book. 148

Table 3.6 Java's eight primitive types, and the Python type that plays each role. 150
Table 3.7 Common operators, from highest precedence to lowest. In Python, `**` binds more tightly than unary minus, so `-2 ** 2` is `-4` 155
Table 3.8 Division and rounding in Java and Python, side by side. 161
Table 3.9 Tracing the `else if` chain of **Listing 3.33** for $k = 2, \ldots, 6$ 164
Table 3.10 The loop forms of Java and Python, and when to use each. 173
Table 3.11 Tracing **Listing 3.43** for $n = 4071$; the loop ends when n reaches 0. 174
Table 3.12 The imperial units of mass and their value in pounds. 177
Table 3.13 Evaluating Equation (3.8) by hand for the first five years. 182
Table 3.14 Checking the four prices of **Listing 3.55** with Equation (3.9) and Equation (3.10) 193
Table 3.15 Values of the expressions in Exercise 3.5. 200
Table 3.16 Tracing the loop of Exercise 3.6 for $x = 48$, $y = 18$. 200
Table 4.1 The fourteen problems of this chapter and the tools from Chapters 1 to 3 that each one exercises most. 207
Table 4.2 The six test times worked by hand: hour hand, minute hand, their difference, the difference brought into 0,360, and the rounded result. 216
Table 4.3 The passwords of two accounts after each step of the test; the last row shows that each object has its own fields. 223
Table 4.4 The fight of Problem 4.7 worked by hand: each attack costs the defender (attacker's power − defender's defense). 246
Table 4.5 Grouping the digits of 1250000 from the right: a dot is added after every third digit, except after the leftmost one. 254
Table 4.6 Trace of the digit-sum loop for $n = 4071$: four passes, one per digit, ending when n reaches 0. 286
Table 4.7 Trace of Euclid's algorithm for $a = 84$, $b = 36$: after two passes $b = 0$ and the answer is $a = 12$ 286
Table 4.8 The fourteen problems of this chapter, the design idea each one teaches, and the trap its tempting wrong solution demonstrates. 298
Table 5.1 The everyday operations on a Java array, a Java `ArrayList`, and a Python `list`. 317
Table 5.2 Choosing between an array and a list. 320
Table 6.1 The Javadoc tags used most often, and where the same information goes in a Python docstring. 393
Table 6.2 The same testing ideas in JUnit 6, `unittest` and pytest. 409
Table 6.3 Equivalence classes and boundary values for the per-kilogram price with a 3 kg minimum. 410
Table 6.4 Overloading in Java and the Python tools that do the same jobs. 429
Table 6.5 Interfaces from the Java library that appear early, and what plays the same part in Python. 446
Table 6.6 Four ways to state "every class of this kind must provide these methods". 458
Table 6.7 The five price rules of Profitina Laundry. 471
Table 7.1 Overloading and overriding compared. 537
Table 7.2 What is decided when: only overridable instance methods are dispatched on the dynamic type. 545
Table 7.3 Abstract classes and interfaces in Java. 571
Table 7.4 Java access levels. *Class*: code in the same class; *Package*: code in any class of the same package; *Subclass*: code in a subclass declared in another package; *World*: any other code. 576
Table 7.5 Visibility of `Alpha`'s members from each class of **Figure 7.10**. 577
Table 7.6 Access control in Java and its conventional counterparts in Python. 581
Table 8.1 The eight-step method for a multi-class design problem. Steps 1 to 6 are design, Step 7 is specification, and Step 8 is implementation. 617
Table 8.2 The nine constructor calls of Problem 8.1 in bottom-up order. A node can be built only after both of its subtrees exist. 631
Table 8.3 The delegation recipe applied to `triple()` 636
Table 8.4 The seven calls of `printDepthFirst()` on the sample tree, in the order they happen. Reading the middle column from top to bottom gives the output. 641

Table 8.5 The delegation recipe for a deep copy. Unlike `triple()`, the results of the delegated calls are the whole point. 646
Table 8.6 The calls of `saveDepthFirst` on the sample tree starting at position 4. Each call returns the last position it wrote; the caller adds one to find the next free cell. 652
Table 8.7 The cases of `find(n)` in a BST. Exactly one of the two subtrees is ever asked. 657
Table 8.8 `insert` is `find` with a different last case: where the search would give up, the new leaf is attached. 662
Table 8.9 Design of `negateAll`: a *map* over the list, one output element per input element. 668
Table 8.10 Design of `find`: a position found in the tail is shifted by one; the "not found" signal is not. 671
Table 8.11 Design of `allPositive`: a logical AND over all elements, with `true` as the neutral starting value. 674
Table 8.12 Design of `positives`: a filter keeps order because each kept head is placed in front of the filtered tail. 677
Table 8.13 Design of `sorted`: each call compares one neighboring pair and delegates the rest. 680
Table 8.14 Design of `merge`: each call moves the smaller head to the result, so each call removes one element from one of the lists. 683
Table 8.15 Design of `skip`: a filter whose test is "not equal to `x`". 686
Table 8.16 Design of `removeDuplicates`: the head survives, its later copies are skipped, and the recursion handles the rest. 689
Table 8.17 Design of the order parser: each kind of failure is reported by the method that detects it and handled by the method that can do something about it. 693
Table 8.18 Design of `GrowableIntArray`. The precondition of `get` is checked against `size`, not against the array length. 701
Table 8.19 How the self-checking programs of this chapter map onto JUnit 6 and `unittest`. 713
Table 8.20 Hand trace of `ServiceDispatch.main`. Overloads are chosen from the declared type; overridden methods from the object's class. 718
Table 8.21 The design problems of the duplicated `Player` and `Enemy` classes and how the refactoring fixes each. 722
Table 8.22 The four encapsulation bugs of `LeakyOrder`. Every one is a path to the state that bypasses a check. 730
Table 8.23 The twenty-one problems of the chapter: what each practices and the single case its tempting wrong solution gets wrong. 737
Table 9.1 The three families of collections compared. 762
Table 9.2 The collections of this chapter in Java and in Python. 763
Table 9.3 What each kind of list declaration may hold, and what it lets you read and write. 810
Table 9.4 Generic programming in Java and in Python. 819
Table 9.5 Cost of the main operations of Java's collection classes for n elements. 831
Table 9.6 Cost of the main operations of Python's collections for n elements. 832
Table 10.1 Common components of Swing and their Tkinter counterparts. 869
Table 10.2 The four major event types: how to listen to them in Swing and in Tkinter. 889
Table 10.3 Swing layout managers and Tkinter geometry managers for common needs. 900
Table 10.4 Ways to write a listener or callback, and what each one remembers. 917
Table 10.5 The thread rule's tools in Swing and in Tkinter. 930
Table 11.1 The responsibilities and permitted dependencies of the three roles. 974
Table 11.2 The Observer pattern in the two languages. 988
Table 11.3 What each key of the keypad does to the message model. 1000
Table 12.1 Six steps for a problem that combines collections, interfaces and a user interface. Steps 1 to 3 choose the data structure, Steps 4 and 5 shape the GUI, and Step 6 checks the choice. 1076
Table 12.2 The four parts of the predictive-text project and the chapters whose ideas each part uses. 1078
Table 12.3 The loop of `wordToSignature("Home")`, one character per step. The buffer grows by one digit at every step and is turned into a `String` once, at the end. 1079
Table 12.4 Binary search for 4663 among the 93 sorted cards of `words.txt`. Four looks find one match; the others lie at indices 38 to 45, around it. 1093

Table 12.5 Elapsed time of each command-line program for the 53 signatures of `sample_text.txt` with the 200,000-word dictionary, as reported by `time`. Every run includes starting the program and building its dictionary. 1112
Table 12.6 The delegation recipe of Chapter 8 applied to the two recursive methods of the tree dictionary. 1120
Table 12.7 The four dictionaries compared. n is the number of words, m their average length, k the number of matches, ℓ the length of the signature and Ws the number of words stored at its node. 1132
Table 12.8 The model's three fields (finished words, candidates, index of the candidate on show) and what each key does to them. No signature is ever stored. 1133
Table 12.9 Three jobs of the pickup counter and the collections that could do them. The costs are per operation unless stated otherwise. 1150
Table 12.10 The six sample orders in board order, with the sort key of each. Orders 105 and 102 are tied on the first two keys and separated by the customer's name. 1162
Table 12.11 The sixteen problems of the chapter: what each practices and the case its tempting wrong solution gets wrong. 1188
Table 13.1 The iteration protocol in Java and Python. 1206
Table 13.2 Safe ways to change a collection that is being walked. 1224
Table 13.3 Tools for immutability in Java and Python. 1247
Table 13.4 Stream operations in Java and their usual Python equivalents. 1264
Table 14.1 The basic thread vocabulary of Java and Python. 1309
Table 14.2 Synchronization tools of Java and Python used in this chapter. 1331
Table 14.3 Ways to prevent deadlock, and the Coffman condition each one breaks. 1354
Table 15.1 TCP and UDP compared. 1388
Table 15.2 The TCP socket calls in Java and Python, in the order a program uses them. 1392
Table 15.3 The knock-knock protocol as a table of states and messages. Matching ignores upper and lower case. 1401
Table 15.4 The order-status protocol. One JSON object per line in each direction; the client speaks first; every request receives exactly one response. 1420
Table 15.5 Common HTTP status codes. Codes 2xx mean success, 4xx a problem with the request, 5xx a problem in the server. 1456
Table 15.6 The main obligations of UU PDP No. 27/2022, and how each appears in a small networked service. 1462
Table 15.7 The Git commands a team member needs in the first week. 1472
Table 15.8 The milestone plan of the sample project, one deliverable per week, each checked by someone other than the person who produced it. 1475
Table 16.1 The tools added by Chapters 13 to 15, in both languages. Every problem of this chapter uses several of them. 1500
Table 16.2 The ten-step method for a team project, and the problem of this chapter in which each step is carried out. 1502
Table 16.3 How the demands of the project brief are divided among Problems 16.1 to 16.8. 1503
Table 16.4 The specification and job description of each part of the Knock-Knock Club. With three members, members 2 and 3 own two parts each. 1504
Table 16.5 The transition table of the knock-knock protocol. Every state has a row for "anything else", which the flow chart does not show. 1514
Table 16.6 What each side observes when the other side fails, in Java and in Python, and how the Knock-Knock Club reacts. 1548
Table 16.7 The product documentation of the Knock-Knock Club, one row per document. 1572
Table 16.8 Troubleshooting the Knock-Knock Club: symptoms as the user sees them, causes, and remedies. 1572
Table 16.9 The LAUNDRY/1 protocol. Command words are case-insensitive; every request line gets exactly one reply line. 1609
Table 16.10 Where the ideas of the whole book appear in the capstone. 1620
Table 16.11 The fourteen problems of the chapter: what each practices and the single case its tempting wrong solution gets wrong. 1645

Table 16.12 The whole book in one table: each chapter's question and where its answer is used in this chapter. 1646
Table 16.13 Design patterns used in this book, by name. 1647

List of Listings

Listing 1.1 `LaundryLedger.java`: the laundry notebook as four parallel arrays and two stand-alone functions, and two careless lines that damage it. 9
Listing 1.2 `laundry_ledger.py`: the same notebook as four parallel Python lists. 9
Listing 1.3 `Hello.java`: the first Java program, which greets the reader and reports which Java version is running it. 24
Listing 1.4 `hello.py`: the first Python program, which does the same as **Listing 1.3** 25
Listing 1.5 `Loads.java`: counting the machine loads for today's orders, a small program with a loop to watch in the debugger. 26
Listing 1.6 `loads.py`: the same count in Python; `//` is Python's whole-number division. 27
Listing 1.7 `Receipt.java`: a receipt computed with variables of four types. 29
Listing 1.8 `receipt.py`: the same receipt in Python, where the values decide their own types. 29
Listing 1.9 `Order.java`: a laundry order that holds its own state, computes its own price, and moves its status forward only when that makes sense. 30
Listing 1.10 `order.py`: the same `Order` class in Python. 32
Listing 1.11 `OrderDemo.java`: creating two orders and sending them messages, including a premature pick-up that is refused. 34
Listing 1.12 `order_demo.py`: the same program in Python, importing the class from `order.py`. 34
Listing 1.13 `ShopDay.java`: a day at Profitina Laundry, using the unchanged `Order` class; compile it together with `Order.java`. 38
Listing 1.14 `shop_day.py`: the same day in Python; it imports `Order` from `order.py` in the same folder. 39
Listing 1.15 `WashingMachine.java`: a washing machine that refuses overloads, loading while running, and starting while empty. 45
Listing 1.16 `washing_machine.py`: the same class in Python; the block under `if __name__ == "__main__":` runs only when the file is run directly, not when it is imported. 46
Listing 2.1 `Add.java`: a two-line method, small enough to read its bytecode. 54
Listing 2.2 `add_bytecode.py`: running a function, then printing the bytecode Python made for it. 55
Listing 2.3 `MyFirstClass.java` in package `myfirstproject`: the first program of the Eclipse project. 68
Listing 2.4 `my_first_module.py`: the same first program in Python. 69
Listing 2.5 `Counter.java` in package `util`: adds $1 + 2 + \cdots + x$ for $1 \le x \le 255$. 75
Listing 2.6 `Tester.java` in package `main`: a small hand-made test of `Counter`. 77
Listing 2.7 `util/counter.py`: the Python twin of `Counter.java`. 78
Listing 2.8 `main/tester.py`: the Python twin of `Tester.java` 78
Listing 2.9 `LaundryPrice.java`: the cashier's price calculator, with a bug. 92
Listing 2.10 `laundry_price.py`: the same calculator in Python, with the same bug. 93
Listing 2.11 `LaundryPriceFixed.java`: the express surcharge now applies to the item's own price. 102
Listing 2.12 `laundry_price_fixed.py`: the same fix in Python. 103
Listing 2.13 `DailyOrders.java`: a day's orders, Rp 6,000 short. 105
Listing 2.14 `daily_orders.py`: the same day's orders in Python, with the same shortfall. 106
Listing 2.15 `UnknownService.java`: an order naming a service the price list does not know. 109
Listing 2.16 `unknown_service.py`: the same order in Python. 109
Listing 2.17 `CounterTwice.java`: a field keeps its value between calls; a local variable starts fresh. 115

Listing 2.18 `counter_twice.py`: the same contrast in Python, with an attribute and a local variable. 115
Listing 2.19 `LaundryReceipt.java`: an itemized receipt with dry cleaning added. 116
Listing 2.20 `laundry_receipt.py`: the same receipt in Python; the `%` formatting uses the same codes as Java's `printf`. 117
Listing 3.1 `Hello.java`: the smallest complete Java program. 121
Listing 3.2 `hello.py`: the same program in Python, with the conventional `main` function. 121
Listing 3.3 `Hello2.java`: printing the first word typed on the command line. 122
Listing 3.4 `hello2.py`: the command-line words live in `sys.argv`, after the script name. 123
Listing 3.5 `CommentDemo.java`: a Javadoc comment, a line comment and a block comment. 124
Listing 3.6 `comment_demo.py`: a docstring and line comments in Python. 124
Listing 3.7 `Dog.java`: four private fields, one constructor, a getter and a setter for each field, and `toString` 126
Listing 3.8 `dog.py`: the class `Dog` in Python; fields are created inside `__init__`. 129
Listing 3.9 `DogDemo.java`: creating two `Dog` objects, printing them, and changing them through setters. 132
Listing 3.10 `dog_demo.py`: the same demonstration in Python. 132
Listing 3.11 `DogCheck.java`: a hand-made test of `getAge` and `setWeight`. 133
Listing 3.12 `dog_check.py`: the same check in Python. 134
Listing 3.13 `ReferenceDemo.java`: copying a primitive copies the value; copying a reference copies the arrow. 137
Listing 3.14 `reference_demo.py`: in Python every variable is a name bound to an object. 139
Listing 3.15 `StringBasics.java`: `startsWith`, `substring`, `charAt`, `length` and other everyday `String` methods. 141
Listing 3.16 `string_basics.py`: the same operations with Python's `str`. 143
Listing 3.17 `StringImmutability.java`: string methods return new strings; `==` compares arrows while `equals` compares characters. 144
Listing 3.18 `string_immutability.py`: Python strings are immutable too; `==` compares content and `is` compares identity. 146
Listing 3.19 `StringFormatting.java`: `printf` and `String.format` with widths, decimals and thousands separators. 147
Listing 3.20 `string_formatting.py`: the same formatting with Python's f-strings and `str.format`. 147
Listing 3.21 `ParseNumbers.java`: turning command-line text into an `int` and a `double`. 148
Listing 3.22 `parse_numbers.py`: `int(...)` and `float(...)` parse text in Python. 149
Listing 3.23 `StaticTyping.java`: in Java every variable has one type for life; `var` lets the compiler work the type out. 151
Listing 3.24 `dynamic_typing.py`: in Python a name can refer to values of different types; types are checked when an operation runs. 152
Listing 3.25 `CharBoolean.java`: a `char` is a small number underneath; `&&`, `||` and `!` combine truth values. 152
Listing 3.26 `char_boolean.py`: `ord` and `chr` convert between characters and codes; `and`, `or` and `not` combine truth values. 153
Listing 3.27 `IntegerTypes.java`: the ranges of the integer types, and an overflow in the middle of a calculation. 155
Listing 3.28 `integer_types.py`: Python integers never overflow; `wrap_int32` imitates Java's 32-bit `int`. 156
Listing 3.29 `FloatingPoint.java`: adding 0.1 ten times, and why decimal numbers must not be compared with `==`. 157
Listing 3.30 `floating_point.py`: the same effects in Python, with `math.isclose` and the `decimal` module 159
Listing 3.31 `IntegerDivision.java`: integer division, the remainder operator, and casting. 160
Listing 3.32 `integer_division.py`: Python separates true division `/` from floor division `//`. 160
Listing 3.33 `IfDemo.java`: an `if`/`else`, and an `else if` chain tried for several values 162

Listing 3.34 `if_demo.py`: `if`, `elif` and `else` in Python. 163
Listing 3.35 `SwitchDemo.java`: fall-through without `break`, the fix with `break` and `default`, and the arrow form. 165
Listing 3.36 `switch_demo.py`: `match` (Python 3.10 and later), an `elif` chain, and a dictionary lookup. 166
Listing 3.37 `LoopBasics.java`: a corrected `while` loop, the equivalent `for` loop, and a `do-while` loop. 168
Listing 3.38 `loop_basics.py`: Python's `while` loop, a `for` loop over `range`, and the `while True` pattern that replaces `do-while`. 169
Listing 3.39 `LoopSequences.java`: four `for` loops, one for each sequence. 170
Listing 3.40 `loop_sequences.py`: the same sequences with `range`. 171
Listing 3.41 `ArgsEcho.java`: visiting every command-line word by index and with a for-each loop. 172
Listing 3.42 `args_echo.py`: the same in Python, with `enumerate` for index and value together. 172
Listing 3.43 `TraceDemo.java`: a `while` loop that takes a number apart digit by digit. 174
Listing 3.44 `trace_demo.py`: the same loop in Python, with `//` for integer division. 174
Listing 3.45 `CircleArea.java`: the area of a circle with radius 5. 175
Listing 3.46 `circle_area.py`: the same computation in Python. 176
Listing 3.47 `ImperialToKg.java`: converting 11 stones and 6 pounds to kilograms. 177
Listing 3.48 `imperial_to_kg.py`: the same conversion in Python. 178
Listing 3.49 `Investment.java`: Equation (3.8) evaluated for six numbers of years, one after another. 180
Listing 3.50 `investment.py`: the same repetitive solution in Python. 181
Listing 3.51 `InvestmentLoop.java`: the balance computed year by year in a loop. 182
Listing 3.52 `investment_loop.py`: the loop version in Python. 183
Listing 3.53 `Customer.java`: a laundry customer with a masked phone number and a visit counter. 185
Listing 3.54 `Service.java`: a laundry service that computes the chargeable weight, the price, and the turnaround time. 186
Listing 3.55 `LaundryDemo.java`: pricing four bags of laundry for two customers. 188
Listing 3.56 `customer.py`: the `Customer` class in Python. 189
Listing 3.57 `service.py`: the `Service` class in Python. 190
Listing 3.58 `laundry_demo.py`: the demonstration program in Python. 191
Listing 3.59 `WashingMachine.java`: solution to Exercise 3.3. 196
Listing 3.60 `washing_machine.py`: solution to Exercise 3.3 in Python. 197
Listing 3.61 `Initials.java`: solution to Exercise 3.4. 198
Listing 3.62 `initials.py`: solution to Exercise 3.4 in Python. 199
Listing 3.63 `MultiplicationTable.java`: solution to Exercise 3.7. 200
Listing 3.64 `multiplication_table.py`: solution to Exercise 3.7 in Python. 201
Listing 3.65 `DoublingYears.java`: solution to Exercise 3.8. 201
Listing 3.66 `doubling_years.py`: solution to Exercise 3.8 in Python. 202
Listing 3.67 `PriceTable.java`: solution to Exercise 3.9. 202
Listing 3.68 `price_table.py`: solution to Exercise 3.9 in Python. 203
Listing 3.69 `BanknoteBreakdown.java`: solution to Exercise 3.10. 204
Listing 3.70 `banknote_breakdown.py`: solution to Exercise 3.10 in Python. 204
Listing 4.1 `Q11Fraction.java`: the sum and product of two fractions with whole numbers only. 210
Listing 4.2 `Q11FractionTest.java`: a plain test driver for Problem 4.1. 211
Listing 4.3 `q11_fraction.py`: Problem 4.1 in Python, with its `unittest` tests. 212
Listing 4.4 `WrongFractionSum.java`: three tempting ways to add 12 and 13. 214
Listing 4.5 `wrong_fraction_sum.py`: the same three mistakes in Python. 215
Listing 4.6 `Q12Time.java`: the counterclockwise angle from the hour hand to the minute hand. 218
Listing 4.7 `Q12TimeTest.java`: the six given times and three edge cases. 219
Listing 4.8 `q12_time.py`: Problem 4.2 in Python, where `%` and `round` already behave as Equation (4.5) requires. 219

Listing 4.9 `WrongClockAngle.java`: three tempting slips in the clock angle at 4:41. 221
Listing 4.10 `wrong_clock_angle.py`: in Python, rounding before normalizing happens to give the right answer. 222
Listing 4.11 `Q13ComputerAccount.java`: an account with private fields and a controlled way to check the password. 223
Listing 4.12 `Q13ComputerAccountTest.java`: the story of **Table 4.3** as automatic checks. 224
Listing 4.13 `q13_computer_account.py`: Problem 4.3 in Python. 225
Listing 4.14 `WrongPasswordCheck.java`: `==` compares object identity, `equals` compares characters. 228
Listing 4.15 `wrong_password_check.py`: in Python, `is` is the identity test and `==` compares contents. 228
Listing 4.16 `Q14ComplexNumber.java`: complex numbers whose operations return new objects. 230
Listing 4.17 `Q14ComplexNumberTest.java`: the four hand examples and a check that the operands are unchanged. 230
Listing 4.18 `q14_complex_number.py`: Problem 4.4 in Python, where `__str__` plays the role of `toString`. 231
Listing 4.19 `WrongComplex.java`: multiplying part by part, and printing a negative part naively. 234
Listing 4.20 `wrong_complex.py`: the same two slips in Python. 234
Listing 4.21 `Q15Weight.java`: a weight stored in pounds, with kilograms computed on demand. 235
Listing 4.22 `Q15WeightTest.java`: floating-point results are compared with a tolerance. 235
Listing 4.23 `q15_weight.py`: Problem 4.5 in Python, with `assertAlmostEqual` for the floating-point checks. 236
Listing 4.24 `WrongWeight.java`: two fields that can fall out of step. 238
Listing 4.25 `wrong_weight.py`: the same stale second value in Python. 238
Listing 4.26 `Q16Fraction.java`: an immutable fraction whose operations return new fractions. 240
Listing 4.27 `Q16FractionTest.java`: the three required outputs, immutability, and a chained computation. 241
Listing 4.28 `q16_fraction.py`: Problem 4.6 in Python. 242
Listing 4.29 `WrongFraction.java`: a `getSum` that changes the object it is called on. 244
Listing 4.30 `wrong_fraction.py`: the same in-place sum in Python. 245
Listing 4.31 `Player.java`: one side of the fight; `Enemy.java` is its mirror image. 247
Listing 4.32 `EnemyPlayerTest.java`: the fight of **Table 4.4**, with a `while` loop for the last rounds. 249
Listing 4.33 `player_enemy.py`: both classes of Problem 4.7 in one Python module. 250
Listing 4.34 `Cheater.java`: code outside `Enemy` tries to write its private field. 253
Listing 4.35 `cheater.py`: Python does not enforce privacy, so the rule lives in the programmer's discipline. 253
Listing 4.36 `Rupiah.java`: exact money in whole rupiah, formatted with dots from the right. 255
Listing 4.37 `RupiahTest.java`: formatting at the group boundaries, and exact arithmetic. 256
Listing 4.38 `rupiah.py`: Problem 4.8 in Python. 257
Listing 4.39 `WrongRupiahFormat.java`: grouping from the left, and money in a `double`. 259
Listing 4.40 `wrong_rupiah_format.py`: the same two mistakes in Python. 260
Listing 4.41 `TimeOfDay.java`: a time of day stored as minutes since midnight. 261
Listing 4.42 `TimeOfDayTest.java`: carry, wrap-around in both directions, padding and comparison. 262
Listing 4.43 `time_of_day.py`: Problem 4.9 in Python. 263
Listing 4.44 `WrongTime.java`: adding minutes without carry, without wrap-around, and without padding. 265
Listing 4.45 `wrong_time.py`: the same three slips in Python. 265
Listing 4.46 `LaundryReceipt.java`: a receipt that prices services with a `switch` and totals them in `Rupiah`. 267
Listing 4.47 `LaundryReceiptTest.java`: Sari's and Budi's receipts and the refusals. 269
Listing 4.48 `laundry_receipt.py`: Problem 4.10 in Python, importing `Rupiah` from `rupiah.py`. 270

Listing 4.49 `WrongReceipt.java`: truncating or rounding the weight, and formatting with an Indonesian locale. 273
Listing 4.50 `wrong_receipt.py`: truncation, rounding, and Python's round-half-to-even. 273
Listing 4.51 `PersonName.java`: word starts found character by character. 275
Listing 4.52 `PersonNameTest.java`: the messy name and three edge cases. 275
Listing 4.53 `person_name.py`: Problem 4.11 in Python. 276
Listing 4.54 `WrongInitials.java`: `split(" ")` produces empty words, and `charAt(0)` of an empty word crashes. 278
Listing 4.55 `wrong_initials.py`: the same empty words in Python, and an `IndexError`. 279
Listing 4.56 `PasswordChecker.java`: count in one loop, decide after it. 280
Listing 4.57 `PasswordCheckerTest.java`: digits at either end, the length boundary, and the empty password. 281
Listing 4.58 `password_checker.py`: Problem 4.12 in Python. 282
Listing 4.59 `WrongPassword.java`: a loop that returns a verdict after the first character. 284
Listing 4.60 `wrong_password.py`: the same early return in Python. 285
Listing 4.61 `LoopTrace.java`: two loops that print their own trace tables. 286
Listing 4.62 `LoopTraceTest.java`: zero, one digit, trailing zeros, and both orders for Euclid. 287
Listing 4.63 `loop_trace.py`: Problem 4.13 in Python, with `//` for integer division. 288
Listing 4.64 `WrongLoop.java`: the condition `n > 9` stops one pass too early. 290
Listing 4.65 `wrong_loop.py`: the off-by-one condition, and `/` where `//` was meant. 290
Listing 4.66 `SimpleDate.java`: leap years by Equation (4.9), month lengths by `switch`, and the next day in three cases. 292
Listing 4.67 `SimpleDateTest.java`: the four kinds of year, month ends, the year end, and invalid dates. 293
Listing 4.68 `simple_date.py`: Problem 4.14 in Python, with `match` in place of `switch`. 294
Listing 4.69 `WrongLeapYear.java`: "divisible by 4" alone. 296
Listing 4.70 `wrong_leap_year.py`: the same Julian rule in Python. 297
Listing 5.1 `FruitArray.java`: creating, filling, and traversing a Java array, plus the shorthand initializer. 306
Listing 5.2 `fruit_array.py`: the same program with a Python list of fixed length. 307
Listing 5.3 `ArrayBounds.java`: a loop that runs one step too far and writes past the end of a five-element array. 308
Listing 5.4 `array_bounds.py`: the same off-by-one error in Python. 309
Listing 5.5 `LoadStats.java`: totals, maximum and average of a `double[]`, and row sums of an `int[][]`. 310
Listing 5.6 `load_stats.py`: the same statistics with a Python list and a list of lists. 311
Listing 5.7 `ArrayCopy.java`: aliasing, copying with `Arrays.copyOf` and `System.arraycopy`, and comparing with `==` and `Arrays.equals` 311
Listing 5.8 `array_copy.py`: aliasing and copying Python lists, and `is` versus `==`. 312
Listing 5.9 `Hello2.java`: printing the command-line arguments. 313
Listing 5.10 `hello2.py`: printing the command-line arguments from `sys.argv`. 314
Listing 5.11 `FruitList.java`: the everyday `ArrayList` operations. 314
Listing 5.12 `fruit_list.py`: the same operations on a Python list. 316
Listing 5.13 `BoxingDemo.java`: boxing, unboxing, and the two meanings of `remove` on a list of `Integer`. 318
Listing 5.14 `boxing_demo.py`: removing by position and by value in Python. 319
Listing 5.15 `SquareInput.java`: reading a line, an integer, a decimal number and another line with `Scanner`. 320
Listing 5.16 `square_input.py`: the same dialogue with `input()`, `int()` and `float()`. 321
Listing 5.17 `FruitCollector.java`: reading lines into an `ArrayList` until the sentinel `#stop#` 322
Listing 5.18 `fruit_collector.py`: the same sentinel loop in Python. 323
Listing 5.19 `InputTest.java`: reading integers until 99, with no protection against bad input. 324

Listing 5.20 `input_test.py`: the same unprotected loop in Python. 325
Listing 5.21 `PropagationDemo.java`: an exception thrown three calls deep and caught in `main` 325
Listing 5.22 `propagation_demo.py`: the same propagation in Python. 327
Listing 5.23 `SafeInput.java`: catching `InputMismatchException` so that bad input is reported and skipped. 330
Listing 5.24 `safe_input.py`: catching `ValueError` from `int()` in Python. 331
Listing 5.25 `CatchOrder.java`: three `catch` clauses, from the most specific to the most general. 332
Listing 5.26 `catch_order.py`: three `except` clauses, from the most specific to the most general. 333
Listing 5.27 `FinallyDemo.java`: `finally` runs after a normal `return` and when an exception escapes. 334
Listing 5.28 `finally_demo.py`: `finally` in Python, and the `else` clause. 335
Listing 5.29 `IllegalArgumentExceptionExample.java`: throwing `IllegalArgumentException` for an out-of-range grayscale value, and recovering by clamping it to 255. 336
Listing 5.30 `grayscale.py`: raising `ValueError` for an out-of-range grayscale value. 337
Listing 5.31 `UncheckedDemo.java`: three unchecked exceptions caused by programming mistakes. 338
Listing 5.32 `unchecked_demo.py`: the same three kinds of mistake in Python. 339
Listing 5.33 `OpenFile.java`: `firstLine` declares `IOException` with `throws`; `main` catches it. 341
Listing 5.34 `open_file.py`: `first_line` documents the exception in its docstring; `main` catches it. 342
Listing 5.35 `WeightCheck.java`: a custom checked exception that carries the offending text and, where there is one, its cause. 343
Listing 5.36 `weight_check.py`: a custom exception class in Python, chained with `raise ... from` 345
Listing 5.37 `ReadUntilStop.java`: reading lines from the keyboard with a `BufferedReader` until `stop` or the end of input. 346
Listing 5.38 `read_until_stop.py`: reading lines from standard input until `stop` or the end of input. 347
Listing 5.39 `CopyFileClassic.java`: copying a file and closing both streams in `finally`, the pre-Java-7 way. 348
Listing 5.40 `copy_file_classic.py`: the same copy with explicit `close()` calls in `finally`. 349
Listing 5.41 `CopyFile.java`: copying with `try`-with-resources, which closes both streams automatically. 349
Listing 5.42 `copy_file.py`: copying with the `with` statement. 350
Listing 5.43 `ReceiptWriter.java`: building text with `StringBuilder`, writing a file, appending to it, and reading it back. 351
Listing 5.44 `receipt_writer.py`: the same receipt with `"".join`, `open` modes `"w"` and `"a"`, and a `for` loop over the file. 353
Listing 5.45 `CsvSplit.java`: the two traps of splitting a CSV line in Java. 354
Listing 5.46 `csv_split.py`: splitting by hand, and splitting correctly with the `csv` module. 355
Listing 5.47 `Order.java`: one laundry order, with its price and its CSV form. 357
Listing 5.48 `order.py`: the same class in Python, with a dictionary of rates. 359
Listing 5.49 `OrderFormatException.java`: a checked exception for one invalid line. 359
Listing 5.50 `order_errors.py`: the same exception in Python. 360
Listing 5.51 `OrderCsv.java`: parsing and validating one line, loading a whole file, and saving. 360
Listing 5.52 `order_csv.py`: the same loading, validation and saving in Python. 362
Listing 5.53 `LaundryApp.java`: loading, reporting, pricing, saving and reloading the orders. 364
Listing 5.54 `laundry_app.py`: the Python application. 366
Listing 5.55 `TenthFruit.java`: the tenth fruit, if there is one. 373
Listing 5.56 `tenth_fruit.py`: the same function in Python. 373
Listing 5.57 `GrowArray.java`: growing by one slot versus doubling. 374
Listing 5.58 `grow_array.py`: the same experiment in Python. 374

Listing 5.59 `SomeNumbers.java`: throwing `IllegalArgumentException` and catching it in the caller....376
Listing 5.60 `some_numbers.py`: the same with `ValueError`....376
Listing 5.61 `AverageInput.java`: robust averaging of typed integers....377
Listing 5.62 `average_input.py`: the same in Python....377
Listing 5.63 `PrepaidCard.java`: a custom checked exception for an insufficient balance....378
Listing 5.64 `prepaid_card.py`: the same card in Python....379
Listing 5.65 `ServiceSummary.java`: orders, kilograms and revenue per service....380
Listing 5.66 `service_summary.py`: the same summary in Python....381
Listing 5.67 `WordCount.java`: counting lines, words and characters of a file named on the command line....382
Listing 5.68 `word_count.py`: the same counts in Python....383
Listing 6.1 `BankAccount.java`: a class with two constructors and a documentation comment on every member....387
Listing 6.2 `bank_account.py`: one `__init__` with a default argument does the work of Java's two constructors....390
Listing 6.3 `RandomDemo.java`: the main methods of `java.util.Random`, with a fixed seed....397
Listing 6.4 `random_demo.py`: the same ideas with Python's `random` module....398
Listing 6.5 `Dog.java`: the class we are going to test. It has no `main`; the test programs below use it....399
Listing 6.6 `dog.py`: the same class in Python, with read-only properties....399
Listing 6.7 `DogCheck.java`: testing `Dog` with a hand-written `check` method and no test library....400
Listing 6.8 `dog_check.py`: the same self-checking program in Python....402
Listing 6.9 `DoubleEquality.java`: why `==` is the wrong test for two doubles....403
Listing 6.10 `double_equality.py`: the same experiment, plus `math.isclose`....404
Listing 6.11 `DogTest.java`: the `Dog` tests written for JUnit 6....405
Listing 6.12 `test_dog.py`: the `Dog` tests written for `unittest`....407
Listing 6.13 `IdentityDemo.java`: a class with no `equals`, and a class with `equals` but no `hashCode`....411
Listing 6.14 `identity_demo.py`: the same two mistakes in Python; the second one is caught....414
Listing 6.15 `Customer.java`: equality by phone number, a matching `hashCode`, and a readable `toString`....416
Listing 6.16 `customer.py`: `__eq__`, a matching `__hash__`, and both `__repr__` and `__str__`....419
Listing 6.17 `AverageTwice.java`: two methods with the same name and almost the same body....422
Listing 6.18 `OverloadDemo.java`: which version of an overloaded method does each call choose?....425
Listing 6.19 `average_twice.py`: in Python, the second `def average` replaces the first....427
Listing 6.20 `overload_demo.py`: default arguments, `*args` and `functools.singledispatch`....428
Listing 6.21 `Measurable.java`: an interface with a single method and no code....429
Listing 6.22 `MeasurableDemo.java`: two classes implement `Measurable`, and one `average` method serves both....429
Listing 6.23 `measurable_demo.py`: the `Measurable` protocol; notice that `Account` and `Country` never mention it....432
Listing 6.24 `dbrep/DBClient.java`: a client that keeps one variable per representation and tests a flag at every operation....434
Listing 6.25 `db_client_v1.py`: the same flag-testing client in Python....438
Listing 6.26 `dbiface/DBClient.java`: one variable of the interface type, no flag, and no tests....439
Listing 6.27 `db_client_v2.py`: the interface as an abstract base class, with two shared methods....444
Listing 6.28 `NameSortDemo.java`: a correct `compareTo` and one that trusts `String.compareTo` to return only −1, 0 or 1....447
Listing 6.29 `name_sort.py`: ordering by a tuple key, the same switch bug, and sorting with a `key` function....451
Listing 6.30 `StopButtonDemo.java`: a button that notifies every registered listener through an interface....454

Listing 6.31 `stop_button_demo.py`: the same listener pattern with a Python protocol. 456
Listing 6.32 `Order.java`: a laundry order, validated on construction and identified by its ticket id. 459
Listing 6.33 `order.py`: the same order in Python, with read-only properties. 461
Listing 6.34 `PriceRule.java`: the interface every price rule implements, with a static helper that formats rupiah. 463
Listing 6.35 `price_rule.py`: `PriceRule` as an abstract base class, and the `rupiah` helper. 464
Listing 6.36 `PerKiloRule.java`: price by started kilogram, with a minimum. 465
Listing 6.37 `PerItemRule.java`: price per piece 466
Listing 6.38 `FlatRule.java`: one fixed price. 467
Listing 6.39 `ExpressRule.java`: a decorator that adds 50%, rounding down. 468
Listing 6.40 `MemberDiscountRule.java`: a decorator that takes a percentage off for members, and refuses an impossible percentage. 469
Listing 6.41 `rules.py`: the five price rules in Python. 470
Listing 6.42 `LaundryDemo.java`: choosing a rule once, then using it only through the interface. 472
Listing 6.43 `laundry_demo.py`: the same morning in Python. 474
Listing 6.44 `PriceRuleTest.java`: JUnit 6 tests for every price rule. 475
Listing 6.45 `PriceRuleCheck.java`: the price-rule tests as a plain program. 477
Listing 6.46 `test_price_rules.py`: pytest tests for every price rule, with parametrized boundaries 478
Listing 6.47 `price_rule_check.py`: the price-rule tests as a plain Python program. 481
Listing 6.48 `KilosCharged.java`: a fully documented helper, with the documented examples in `main` 487
Listing 6.49 `kilos_charged.py`: a Google-style docstring whose examples are doctests, one of them an expected exception. 488
Listing 6.50 `ChainedAccount.java`: two public constructors that delegate to one private constructor. 488
Listing 6.51 `chained_account.py`: alternative constructors written as class methods. 489
Listing 6.52 `BankAccountTest.java`: JUnit 6 tests for `interestOn`, comparing doubles with a delta. 490
Listing 6.53 `test_bank_account.py`: `unittest` tests for `interest_on` with `assertAlmostEqual`. 491
Listing 6.54 `MemberCardCheck.java`: the loyalty-card tests, written before the code they test. 492
Listing 6.55 `MemberCard.java`: the loyalty-card rule, written to make the tests pass. 493
Listing 6.56 `test_member_card.py`: the pytest version of the loyalty-card tests. 494
Listing 6.57 `member_card.py`: the rule, with floor division. 494
Listing 6.58 `Machine.java`: equality and hash code built from two fields. 495
Listing 6.59 `machine.py`: `__eq__` and `__hash__` built from a tuple of two fields. 495
Listing 6.60 `CustomerRecordDemo.java`: a record compares all of its components. 496
Listing 6.61 `customer_record.py`: a frozen dataclass compares all of its fields. 496
Listing 6.62 `OverloadPuzzle.java`: overload resolution with widening and boxing. 497
Listing 6.63 `overload_puzzle.py`: `singledispatch` with `bool`, `Fraction` and a big integer. 498
Listing 6.64 `AverageLoad.java`: an adapter lets an `Order` take part in `average` 498
Listing 6.65 `average_load.py`: the same adapter in Python. 499
Listing 6.66 `BulkDiscountRule.java`: a decorator that discounts heavy orders. 500
Listing 6.67 `BulkDiscountCheck.java`: the boundary tests for the bulk discount. 500
Listing 6.68 `bulk_discount.py`: the bulk discount in Python. 501
Listing 6.69 `test_bulk_discount.py`: pytest boundary tests for the bulk discount. 502
Listing 6.70 `DecoratorOrder.java`: stacking the same two decorators in both orders. 502
Listing 6.71 `decorator_order.py`: the same experiment in Python. 503
Listing 6.72 `OrderSortDemo.java`: a `Comparator` that sorts heaviest first, then by ticket id. 504
Listing 6.73 `order_sort_demo.py`: sorting with a tuple key. 504

Listing 7.1 `db/DBTest.java`: the database project with an abstract superclass `DB` and two subclasses, `LinearDB` and `TreeDB`, whose methods only report that they were called. 509
Listing 7.2 `db_test.py`: the database project in Python. `DB` inherits from `ABC` and marks `add_key` and `search` as abstract. 512
Listing 7.3 `IntDBDemo.java`: a working key database. `LinearIntDB` overrides the inherited `addSeveral` with a bulk copy; `TreeIntDB` inherits it unchanged. 514
Listing 7.4 `int_db_demo.py`: the same working database in Python; `LinearIntDB` overrides `add_several` with one call to `list.extend`. 519
Listing 7.5 `ConstructorChain.java`: a three-level chain `LaundryItem`, `Garment`, `Shirt`; each constructor announces itself, and each `describe` extends its parent's text with `super.describe()`. 524
Listing 7.6 `constructor_chain.py`: the same chain in Python, plus a class that forgets to call `super().__init__()`. 526
Listing 7.7 `animal/AnimalTest.java`: `Animal` with overloaded constructors and `getPrice` methods, three subclasses that override `sound()`, and a test program that exercises both overloading and overriding. 529
Listing 7.8 `animal_test.py`: the Animal program in Python; one `__init__` and one `get_price` with optional keyword parameters replace Java's four overloads each. 537
Listing 7.9 `DispatchDemo.java`: dynamic dispatch of `start()`, the non-dispatch of a hidden field and a hidden static method, a safe downcast with `instanceof`, and an unsafe one. 542
Listing 7.10 `dispatch_demo.py`: dynamic lookup in Python, with a helper that walks the method resolution order exactly as Python does. 547
Listing 7.11 `ObjectRoot.java`: a class that inherits `toString`, `equals` and `hashCode` from `Object`, and one that overrides all three. 549
Listing 7.12 `object_root.py`: the same three methods in Python, and a class that defines `__eq__` without `__hash__`. 551
Listing 7.13 `FigureDemo.java`: an abstract `Figure` with abstract subclasses `OpenFigure` and `ClosedFigure` and four concrete figures. 554
Listing 7.14 `figure_demo.py`: the figure hierarchy in Python; `OpenFigure` and `ClosedFigure` list `ABC` again to make clear that they, too, are abstract. 557
Listing 7.15 `dbconcrete/DBTest.java`: `TreeDB` inherits from the concrete class `LinearDB`. 560
Listing 7.16 `db_concrete.py`: the concrete-superclass version in Python. 563
Listing 7.17 `FragileBase.java`: a counting tag set built by inheritance, which counts every tag twice, and one built by composition, which counts correctly. 566
Listing 7.18 `fragile_base.py`: the fragile base class in Python; `UserDict.update` calls `__setitem__` for every pair. 568
Listing 7.19 `DiamondDemo.java`: two interfaces with conflicting default methods; the implementing class must resolve the conflict, and a method inherited from a class beats any default. 571
Listing 7.20 `diamond_mro.py`: a diamond hierarchy with cooperative `super()` calls, the resulting MRO, a conflict resolved by order, and a hierarchy for which no consistent order exists. 573
Listing 7.21 `one/Alpha.java`: one field of each access level. 577
Listing 7.22 `one/Beta.java`: same package as `Alpha`, not a subclass. 577
Listing 7.23 `two/AlphaSub.java`: another package, but a subclass of `Alpha`. 577
Listing 7.24 `two/Gamma.java`: another package, no relationship; contains `main`. 577
Listing 7.25 `access_demo.py`: Python's three naming conventions, and what name mangling really does. 579
Listing 7.26 `EncapsulationDemo.java`: a `WashingMachine` whose private fields can change only through methods that preserve the invariant $0 \le \ell \le c$. 581
Listing 7.27 `encapsulation_demo.py`: the same machine in Python; `capacity_kg` is a read-only property and `load_kg` a property whose setter enforces the invariant. 583
Listing 7.28 `LaundryPricing.java`: the Profitina Laundry service hierarchy, the express decorator, and an order that prices everything polymorphically. 587
Listing 7.29 `laundry_pricing.py`: the same price engine in Python. 593
Listing 7.30 `StudentFees.java`: the student hierarchy of **Figure 7.5** with polymorphic fees. 602

Listing 7.31 `student_fees.py`: the student hierarchy in Python. 605
Listing 7.32 `ShoePromoDemo.java`: shoe cleaning and promotions added to the price engine without modifying it. 607
Listing 7.33 `shoe_promo_demo.py`: the same extension in Python. 609
Listing 7.34 `ConstructorTrap.java`: a superclass constructor that calls an overridable method. 612
Listing 7.35 `constructor_trap.py`: the same trap in Python. 612
Listing 8.1 `TreeStart.java`: the starting `Tree` class with two constructors, `mirror()` by delegation, a compact `toString()` and a sideways drawing. 618
Listing 8.2 `tree_start.py`: the same `Tree` class in Python. The leading underscore marks an attribute as private by convention. 621
Listing 8.3 `List.java`: an immutable list of integers. The constructors are private, so `nil()`, `cons()` and `of()` are the only ways to build a list. 623
Listing 8.4 `ListException.java`: an unchecked exception for operations that need a non-empty list. 625
Listing 8.5 `ListOps.java`: helper operations written recursively. The private constructor stops anyone from creating a `ListOps` object. 625
Listing 8.6 `ListStart.java`: a first tour of `List`, `ListOps` and `ListException`. 627
Listing 8.7 `cons_list.py`: the list class, its exception and the helper operations in one Python module. 627
Listing 8.8 `list_start.py`: a first tour of the Python list module. 629
Listing 8.9 `BuildTree.java`: the nine `new Tree` statements, a recursive `size()`, and self-checks. 631
Listing 8.10 `build_tree.py`: the same tree in Python, tested with `unittest`. 633
Listing 8.11 `TripleTree.java`: tripling every node by delegation, with a crashing wrong variant. 636
Listing 8.12 `triple_tree.py`: tripling every node in Python. 638
Listing 8.13 `DepthFirstPrint.java`: pre-order printing, with a helper that captures console output for testing. 641
Listing 8.14 `depth_first_print.py`: pre-order printing in Python, tested by capturing standard output. 643
Listing 8.15 `FreshCopy.java`: a deep copy, a shallow wrong copy, and a test for shared node objects. 646
Listing 8.16 `fresh_copy.py`: a deep copy in Python, with the same sharing test. 649
Listing 8.17 `SaveDepthFirst.java`: writing the pre-order traversal into an array and returning the last position used. 652
Listing 8.18 `save_depth_first.py`: the same traversal saved into a Python list. 654
Listing 8.19 `BstFind.java`: searching a BST along one path, with a visit counter and a crashing wrong variant. 657
Listing 8.20 `bst_find.py`: the same search in Python. 659
Listing 8.21 `BstInsert.java`: inserting into a BST by attaching a new leaf, with an in-order check. 662
Listing 8.22 `bst_insert.py`: BST insertion in Python. 665
Listing 8.23 `NegateAll.java`: negating every element recursively. 668
Listing 8.24 `negate_all.py`: the same function in Python. 669
Listing 8.25 `ListFind.java`: the position of the first occurrence, or `-1`. 671
Listing 8.26 `list_find.py`: the same search in Python. 673
Listing 8.27 `AllPositive.java`: checking that no element is below zero. 675
Listing 8.28 `all_positive.py`: the same check in Python. 675
Listing 8.29 `Positives.java`: filtering the elements that are at least zero. 677
Listing 8.30 `positives.py`: the same filter in Python. 678
Listing 8.31 `SortedCheck.java`: checking non-decreasing order pair by pair. 680
Listing 8.32 `sorted_check.py`: the same check in Python, named `is_sorted` so as not to hide the built-in `sorted`. 681
Listing 8.33 `MergeLists.java`: merging two sorted lists with both base cases. 683
Listing 8.34 `merge_lists.py`: the same merge in Python. 685

Listing 8.35 `SkipElement.java`: removing every copy of one value. 686
Listing 8.36 `skip_element.py`: the same filter in Python. 687
Listing 8.37 `RemoveDuplicates.java`: removing duplicates with the help of `skip` 689
Listing 8.38 `remove_duplicates.py`: the same method in Python, importing `skip` from Problem 8.14. 690
Listing 8.39 `LaundryOrderParser.java`: a line parser with a custom checked exception, a file reader with try-with-resources, and a wrong reader for comparison. 693
Listing 8.40 `laundry_order_parser.py`: the same parser in Python, with `unittest` tests. 697
Listing 8.41 `GrowableIntArray.java`: a self-doubling array with complete Javadoc and boundary checks. 701
Listing 8.42 `growable_int_array.py`: the same class in Python, with docstrings and a `setUp` method shared by the tests. 705
Listing 8.43 `PriceRules.java`: an interface with a default method, two implementations, and tests that treat both through the interface. 708
Listing 8.44 `price_rules.py`: the same design with an abstract base class, and a reusable contract test run against both implementations. 710
Listing 8.45 `ServiceDispatch.java`: an abstract class, three subclasses, overriding with `super`, and two overloads of `label`. 715
Listing 8.46 `service_dispatch.py`: the same hierarchy in Python, with `singledispatch` choosing by run-time class 719
Listing 8.47 `FighterArena.java`: the refactored hierarchy with a final template method, a protected hook and no public setter. 723
Listing 8.48 `fighter_arena.py`: the same hierarchy in Python, with a read-only `health` property. 727
Listing 8.49 `OrderBugHunt.java`: a class with four encapsulation bugs, demonstrated, and the fixed class, tested. 730
Listing 8.50 `order_bug_hunt.py`: the same hunt in Python, fixed with properties, a validating setter and tuples. 733
Listing 8.51 `TraceFinally.java`: a `try` with `catch` and `finally`, and an exception that the `catch` does not handle. 740
Listing 8.52 `trace_finally.py`: the same flow in Python, where both errors are a `ValueError`. 741
Listing 8.53 `ExerciseSolutions.java`: height, sum, leaves, BST minimum, list sum, take and persistent insertion, with checks. Compile it together with `List.java` and `ListException.java` 742
Listing 8.54 `exercise_solutions.py`: the same solutions in Python. 744
Listing 9.1 `InterfaceConstants.java`: constants in the interfaces `Measurable` and `Units`, used with and without a prefix, and an `enum` as the modern alternative. 750
Listing 9.2 `interface_constants.py`: class attributes marked `Final`, a module constant, and an `Enum`, whose members cannot be reassigned. 753
Listing 9.3 `Fibonacci.java`: a countdown, the standard two-call Fibonacci with a call counter, a tail-recursive version behind a helper method, and a linear version that returns two values at once. 756
Listing 9.4 `fibonacci.py`: the same four functions in Python, plus the standard version made fast by `functools.lru_cache` 758
Listing 9.5 `ListDemo.java`: adding, testing, counting, bulk operations, positions, and one method that serves two list classes. 764
Listing 9.6 `list_demo.py`: the same operations with Python's `list`, and one function that serves both `list` and `deque`. 765
Listing 9.7 `IterationDemo.java`: three ways to visit a list; removing inside a for-each loop fails, while `Iterator.remove` and `removeIf` succeed. 768
Listing 9.8 `iteration_demo.py`: the same walks in Python. Removing from a list inside a `for` loop silently skips elements; changing a dictionary's size inside a loop raises `RuntimeError` 769
Listing 9.9 `SetDemo.java`: a `TreeSet` of numbers with union, difference and intersection; the same names in a `HashSet`, a `LinkedHashSet` and a `TreeSet`; nearest-element queries; and sets of letters. 771

Listing 9.10 `set_demo.py`: Python's `set` with its operators, the orders Java's three sets provide, binary search on a sorted list, and a `frozenset`. 773
Listing 9.11 `MapDemo.java`: walking a map's entries, first with raw types and casts, then with generics; the main `Map` methods on a price list; counting with `merge`; and three kinds of map. 778
Listing 9.12 `map_demo.py`: Python's `dict`, its `get` with a default, `KeyError`, and three ways to count: `get`, `Counter` and `defaultdict`. 780
Listing 9.13 `WordFile.java`: a file read into a `List` of words and a `List` of lines, the words counted in a `TreeMap`, and the most frequent found by sorting the entries. 782
Listing 9.14 `word_file.py`: the same report in Python, with a sort key that orders by count descending and word ascending, and `Counter.most_common`. 784
Listing 9.15 `QueueDemo.java`: an `ArrayDeque` as a FIFO queue, as a LIFO stack and as a deque; a `PriorityQueue` with natural order and with `Comparator.reverseOrder()`. 786
Listing 9.16 `queue_demo.py`: `deque` as a queue and a deque, `list` as a stack, and `heapq` as a priority queue. 788
Listing 9.17 `RawTypes.java`: a raw `List` accepts an `Integer` among the names and fails only when it is cast; the generic `List<String>` needs no cast and would reject the `Integer` at compile time. 790
Listing 9.18 `raw_types.py`: a Python list accepts anything; the hint `list[str]` lets mypy reject the `append(42)` before the program runs. 792
Listing 9.19 `GListDemo.java`: `GList<T>`, a generic immutable list with a recursive `toString` helper, used with `Integer` and with `String`. 793
Listing 9.20 `glist_demo.py`: the generic list with `TypeVar` and `Generic[T]`, checked by mypy. 795
Listing 9.21 `GenericMethods.java`: a generic class with two type parameters, `Pair<A, B>`, and four generic methods, among them `removeDuplicatesFromSortedList`. 797
Listing 9.22 `generic_methods.py`: `Pair(Generic[A, B])` and generic functions typed with `TypeVar`. 800
Listing 9.23 `GList2Demo.java`: a list whose element type is bounded by `Comparable`, with a recursive `sorted` method, and a bounded generic method `max`. 803
Listing 9.24 `glist2_demo.py`: a `TypeVar` bounded by the protocol `SupportsLessThan`; mypy rejects a list of plain `object`s, and Python fails when it tries to compare them. 806
Listing 9.25 `WildcardDemo.java`: `Animal`, `Cat` and `Dog`; a reader method with `? extends Animal`; the method `addAll(src, dest)` with `? super T`; and the covariant array that fails at run time. 810
Listing 9.26 `wildcard_demo.py`: `Sequence[Animal]` accepts a list of cats (read-only, covariant); `list[Animal]` does not (invariant), and when the rule is ignored a dog ends up among the cats. 814
Listing 9.27 `ErasureDemo.java`: at run time a `List<String>` and a `List<Integer>` are the same class; a raw reference can smuggle an `Integer` into a `List<String>`, and `Collections.checkedList` stops it at the door. 816
Listing 9.28 `erasure_demo.py`: `list[str]` is only an alias for `list`; nothing checks type arguments at run time, but the hints stay available to tools. 817
Listing 9.29 `MyCollection.java`: an iterator, the fact that an iterator only reads, removing duplicates from a sorted list, an `Iterable` in a for-each loop, and sorting students with `AgeComparator`. 819
Listing 9.30 `my_collection.py`: the five parts in Python; a Java-style comparison function adapted with `functools.cmp_to_key`, and the Pythonic `key=` function. 823
Listing 9.31 `SortingDemo.java`: natural order, `comparingInt(...).reversed()`, `comparing(...).thenComparing(...)`, a lambda comparator, a `TreeSet` with a comparator, and the overflow in a subtraction comparator. 826
Listing 9.32 `sorting_demo.py`: natural order with `__lt__` and `total_ordering`, key functions, two stable sorts for mixed directions, and Python's unbounded integers. 829
Listing 9.33 `LaundryOrderBook.java`: Profitina Laundry's order book with a `TreeMap`, a generic `MultiIndex`, an `EnumMap` of `LinkedHashSet`s and a `PriorityQueue`, plus revenue reports sorted with comparators. 834

Listing 9.34 `laundry_order_book.py`: the same order book in Python, with a generic `MultiIndex`, dictionaries used as ordered sets, and `heapq` for the wash queue. 842
Listing 9.35 `Anagrams.java`: anagram groups in a `TreeMap` keyed by the sorted letters of each word. 853
Listing 9.36 `anagrams.py`: the same grouping with a `defaultdict`. 854
Listing 9.37 `StackDemo.java`: a generic `ArrayStack<T>` whose top is the end of a private `ArrayList`, and a bracket checker built on it. 855
Listing 9.38 `stack_demo.py`: the generic stack in Python, and the same bracket checker with a partner table. 857
Listing 9.39 `LruCache.java`: an LRU cache in a dozen lines, by extending `LinkedHashMap` in access order 860
Listing 9.40 `lru_cache.py`: the same cache with `OrderedDict` and `move_to_end` 861
Listing 10.1 `FirstWindow.java`: a frame with one label, built in six steps, with a `--selftest` mode that runs without a screen. 870
Listing 10.2 `first_window.py`: the same window in Tkinter, with the same `--selftest` mode. 872
Listing 10.3 `ToyEventLoop.java`: a hand-written event loop with pretend components, listeners and a queue. 876
Listing 10.4 `toy_event_loop.py`: the same event loop in Python, with callbacks as plain functions. 878
Listing 10.5 `EchoPanel.java`: a panel that listens for Enter in its text field and copies the text into a label. 881
Listing 10.6 `echo_panel.py`: the same echo panel in Tkinter, using `bind` to attach a callback to the Enter key. 883
Listing 10.7 `FruitChoice.java`: two check boxes and one item listener that reports which box changed and how. 885
Listing 10.8 `fruit_choice.py`: the same choice in Tkinter, with a `BooleanVar` per check button and a `command` callback. 887
Listing 10.9 `LayoutTour.java`: the four main layout managers; the self-test prints the bounds each manager assigns 892
Listing 10.10 `layout_tour.py`: `pack` and `grid` at work; the self-test prints the geometry Tk assigns. 896
Listing 10.11 `ShadeBox.java`: painting in `paintComponent`, two buttons told apart with `getSource()`, and mouse clicks on the painted box. 902
Listing 10.12 `shade_box.py`: the shading box on a Tkinter `Canvas`, where the box is an item that Tk remembers and redraws. 907
Listing 10.13 `ListenerStyles.java`: five ways to write an action listener in Java, and lambdas created in a loop 910
Listing 10.14 `listener_styles.py`: five kinds of Tkinter callback, the late-binding trap in a loop, and its cure. 914
Listing 10.15 `Doodle.java`: drawing with the mouse, with an anonymous `MouseAdapter` and a painting method that redraws every stored stroke. 918
Listing 10.16 `doodle.py`: drawing with the mouse on a Tkinter canvas; every segment is a line item that Tk keeps. 920
Listing 10.17 `SlowWork.java`: slow work on the EDT freezes the window; a `SwingWorker` runs it on a worker thread and shows the result on the EDT. 923
Listing 10.18 `slow_work.py`: a worker thread computes, a `queue.Queue` carries the result, and `after` polls the queue on the GUI thread. 927
Listing 10.19 `LaundryOrderWindow.java`: the Profitina Laundry order-entry window in Swing, with its model classes and a `--selftest` mode. 934
Listing 10.20 `laundry_order_window.py`: the same order-entry window in Tkinter, with the same model and the same self-test 944
Listing 10.21 `MachineBoard.java`: four machines painted as circles; a click inside a circle, tested with Equation (10.8), toggles the machine. 961
Listing 10.22 `machine_board.py`: the same board on a Tkinter canvas, where `tag_bind` on each oval does the hit test. 964

Listing 11.1 `StudentMvcDemo.java`: the smallest MVC program. `Student` is the model, `StudentView` the view and `StudentController` the controller. 977
Listing 11.2 `student_mvc_demo.py`: the student record in Python, with the model's fields exposed as properties. 979
Listing 11.3 `ObserverDemo.java`: a subject, `WashCounter`, that announces every change of its `loads` property through `PropertyChangeSupport`, and two observers that know nothing of each other. 983
Listing 11.4 `observer_demo.py`: the Observer pattern written by hand. The subject keeps a list of callables and calls each with the property name and the old and new values. 985
Listing 11.5 `CounterMvc.java`: the Up/Down counter as a model, a view interface with a Swing and a text implementation, and a controller. With `--selftest` it runs without a window. 990
Listing 11.6 `counter_mvc.py`: the counter in Python. `TkCounterView` imports Tkinter inside its constructor, so the self-test runs even where Tkinter is not installed. 995
Listing 11.7 `PredictiveKeypad.java`: a phone keypad with predictive text. `T9Dictionary` and `MessageModel` form the model, `KeypadController` the controller, `SwingMessageView` and `SwingMatchView` two views of the same model, and `KeypadPanel` the twelve buttons; `TextMessageView` replaces the window in the self-test. 1000
Listing 11.8 `predictive_keypad.py`: the predictive keypad in Python. The model and controller match the Java classes; `TkKeypadWindow` builds the message box, the match line and the twelve buttons with Tkinter. 1010
Listing 11.9 `MessageModelCheck.java`: ten checks of the keypad model and controller, run without any window. 1017
Listing 11.10 `test_keypad_model.py`: unit tests of the keypad model and controller with `unittest`. 1020
Listing 11.11 `OrderBook.java`: the model of the front desk, with the order book, one order, the two enumerations and the listener interface. It imports nothing from Swing. 1024
Listing 11.12 `order_book.py`: the front-desk model in Python. Like its Java twin, it imports nothing from Tkinter. 1029
Listing 11.13 `LaundryDesk.java`: the view contracts, the mediating controller, the text views of the self-test, and `main`. 1031
Listing 11.14 `DeskWindow.java`: the Swing views of the front desk. `OrderFormPanel`, `OrderTablePanel` and `SummaryBar` implement the three view contracts; `DeskWindow` places them in a frame. 1039
Listing 11.15 `laundry_desk.py`: the view protocols, the controller, the text views of the self-test, and `main`. 1044
Listing 11.16 `tk_desk_views.py`: the Tkinter views of the front desk. 1050
Listing 11.17 `OrderBookCheck.java`: twenty checks of the front-desk model, including prices, refusals, the state machine, the read-only list and unsubscribing. 1053
Listing 11.18 `test_order_book.py`: unit tests of the front-desk model with `unittest`, using `subTest` for the five refusals. 1056
Listing 11.19 `StudentTwoViews.java`: a controller with any number of views; the original `StudentView` and a new comma-separated view. It uses `Student` and `StudentView` from `StudentMvcDemo.java`. 1062
Listing 11.20 `student_two_views.py`: the same two views in Python, importing the model and the first view from `student_mvc_demo.py`. 1064
Listing 11.21 `PickupBoard.java`: a new view of the order book for the waiting-room screen, added without changing the model. Compile it with `OrderBook.java`. 1066
Listing 11.22 `pickup_board.py`: the waiting-room view in Python; an object with `__call__` is a callable, so it can be registered as a listener. 1067
Listing 11.23 `MiniBinding.java`: a generic observable `Property` and a view model of the order form. Compile it with `OrderBook.java`. 1069
Listing 11.24 `mini_binding.py`: the same property and view model in Python. 1071
Listing 12.1 `PredictivePrototype.java`: the signature of a word, the validity test and the file-scanning lookup, all as static methods of one utility class. 1079

Listing 12.2 `PrototypeCheck.java`: checks for Problems 12.1 and 12.2, two wrong variants, and a timing of `String` against `StringBuffer` 1081
Listing 12.3 `prototype.py`: the same three functions in Python. The table `_KEY_OF` is built once, when the module is first imported. 1084
Listing 12.4 `prototype_check.py`: tests for Problems 12.1 and 12.2 in Python, with the same wrong variants and a timing of three ways to build a string. 1085
Listing 12.5 `Words2SigProto.java`: the signature of every word on the command line. 1089
Listing 12.6 `Sigs2WordsProto.java`: the words of every signature on the command line, one line per signature, in alphabetical order. 1090
Listing 12.7 `words2sig_proto.py`: the same program in Python; the words are `sys.argv[1:]`. 1091
Listing 12.8 `sigs2words_proto.py`: the words of every signature on the command line, sorted. 1091
Listing 12.9 `WordSig.java`: a signature and a word, compared in the numerical order of Equation (12.4). 1093
Listing 12.10 `DictionaryListImpl.java`: the dictionary as a sorted `ArrayList<WordSig>`, searched with `Collections.binarySearch` and a walk to both sides. 1094
Listing 12.11 `ListCheck.java`: checks for Problem 12.4, the two orders of signatures side by side, and two wrong variants. 1096
Listing 12.12 `word_sig.py`: the pair class in Python; `__lt__` is all that sorting and `bisect` need. 1098
Listing 12.13 `dictionary_list_impl.py`: the sorted list in Python, searched with `bisect_left` and a walk upward. 1098
Listing 12.14 `list_check.py`: tests for Problem 12.4 in Python, with the same two wrong variants. 1099
Listing 12.15 `Dictionary.java`: one method, documented with Javadoc, including the empty and invalid cases. 1102
Listing 12.16 `dictionary.py`: the same contract as an abstract base class with a docstring. 1103
Listing 12.17 `Sigs2WordsList.java`: the command-line program on top of any `Dictionary`; this one creates a `DictionaryListImpl`. 1104
Listing 12.18 `MakeDictionary.java`: 200,000 pronounceable invented words, the same on every computer, written to `big_words.txt`. 1105
Listing 12.19 `TimingComparison.java`: build time and lookup time of the four dictionaries on the same file and the same 2,000 signatures. 1106
Listing 12.20 `sigs2words_list.py`: the list program in Python. 1108
Listing 12.21 `make_dictionary.py`: the same invented dictionary in Python. 1109
Listing 12.22 `timing_comparison.py`: the four dictionaries measured in Python. 1110
Listing 12.23 `DictionaryMapImpl.java`: a `HashMap` from signatures to sets of words, with a public `addWord` for fast insertion and a read-only result. 1113
Listing 12.24 `MapCheck.java`: checks for Problem 12.7, including an attempt to clear a returned set, and a map class with two wrong methods. 1114
Listing 12.25 `dictionary_map_impl.py`: the multi-valued map in Python, returning a `frozenset`. 1116
Listing 12.26 `map_check.py`: tests for Problem 12.7 in Python, and the same two wrong methods. 1117
Listing 12.27 `DictionaryTreeImpl.java`: a trie with eight children per node, filled and searched by delegation; results are trimmed to the length of the signature. 1121
Listing 12.28 `TreeCheck.java`: checks for Problem 12.8 and a tree that stores words only at the ends of their paths. 1123
Listing 12.29 `dictionary_tree_impl.py`: the same trie in Python; `_Node` does the recursion and `_trim` is a set comprehension. 1125
Listing 12.30 `tree_check.py`: tests for Problem 12.8 in Python, and the end-only tree. 1126
Listing 12.31 `Sigs2WordsMap.java`: the command-line program over a `DictionaryMapImpl` 1129
Listing 12.32 `Sigs2WordsTree.java`: the command-line program over a `DictionaryTreeImpl` 1129
Listing 12.33 `sigs2words_map.py`: the map program in Python. 1130
Listing 12.34 `sigs2words_tree.py`: the tree program in Python. 1131
Listing 12.35 `DictionaryModel.java`: the model. It stores the finished words, the candidates and an index, and recomputes the signature from the word on show. 1135
Listing 12.36 `MessageView.java`: the view, a read-only text area inside a scroll pane. 1137

Listing 12.37 `KeypadController.java`: the controller, twelve buttons in a 4 by 3 grid built from one table of keys and labels. 1138
Listing 12.38 `PredictiveGUI.java`: the main program, with a `--selftest` mode that clicks the buttons from code. 1139
Listing 12.39 `ModelCheck.java`: tests of the model alone, with a fake dictionary written as a lambda. 1141
Listing 12.40 `dictionary_model.py`: the model in Python; `from_file` is an alternative constructor, written as a class method. 1143
Listing 12.41 `message_view.py`: the view, a Tkinter `Text` widget that is unlocked only while the model's text is written into it. 1144
Listing 12.42 `keypad_controller.py`: the controller; `make_command` gives every button a function that remembers its own key. 1145
Listing 12.43 `predictive_gui.py`: the main program; `--selftest` presses the keys through the same `make_command` functions that the buttons use. 1145
Listing 12.44 `model_check.py`: tests of the model alone; the fake dictionary is any object with a `signature_to_words` method. 1147
Listing 12.45 `CollectionChoice.java`: membership tests on three collections, the iteration order of three sets, and two ways of removing duplicates, all timed. 1150
Listing 12.46 `collection_choice.py`: the same experiment in Python; a sorted list with `bisect` stands in for `TreeSet`. 1153
Listing 12.47 `GenericBin.java`: a bounded generic class `Bin<T extends Weighed>`, a wildcard method `totalKg` and a generic method `maxOf`, with checks. 1155
Listing 12.48 `generic_bin.py`: the same design with `Protocol`, `TypeVar` and `Generic`, checked by mypy before the run and by `isinstance` during it. 1158
Listing 12.49 `OrderSorting.java`: the pickup board sorted by a four-key `Comparator` chain, with two tempting wrong comparators. 1163
Listing 12.50 `order_sorting.py`: the same board with a tuple key and with two stable sorts, and the same two wrong orders. 1164
Listing 12.51 `BagCounterApp.java`: a counter in MVC form with two views, a self-test that exercises the model alone and then clicks the real buttons, and a panel without a model for comparison. 1167
Listing 12.52 `bag_counter.py`: the same counter in Python; a view receives the function that displays its text, so tests can replace the label. 1172
Listing 12.53 `ObserverBugs.java`: a subject that notifies before changing and walks its live list of listeners, next to the fixed version. 1176
Listing 12.54 `observer_bugs.py`: the same subject in Python, where the second bug fails silently. 1179
Listing 12.55 `KeypadClosures.java`: listeners made in a loop with a lambda, an anonymous inner class and a named class, and a version whose listeners share one mutable cell. 1182
Listing 12.56 `keypad_closures.py`: commands made in a loop, one version with the late-binding bug and three correct ones. 1185
Listing 12.57 `ExerciseSolutions.java`: a pair with `equals` and `hashCode`, a table-driven signature, a `TreeMap` dictionary with prefix ranges, a generic `invert`, and a comparator, with checks. 1191
Listing 12.58 `exercise_solutions.py`: the same solutions in Python; the prefix search uses two `bisect_left` calls on the sorted signatures. 1195
Listing 13.1 `IteratorBasics.java`: walking a collection with an explicit iterator, changing a local copy of an element, the enhanced `for` loop, and asking for one element too many. 1201
Listing 13.2 `iterator_basics.py`: `iter()`, `next()` and `StopIteration`, the `for` loop, and asking for one element too many. 1204
Listing 13.3 `RemoveDuplicates.java`: removing adjacent duplicates from a sorted list with a remembered previous element, a version that handles empty lists and `null`, and an in-place version with `ListIterator`. 1207
Listing 13.4 `remove_duplicates.py`: the same task in Python, with a flag, with `itertools.groupby`, in place by walking indexes from the end, and walking backwards with `reversed`. 1211

Listing 13.5 `LoadPlanDemo.java`: `LoadPlan` is an `Iterable` that computes its loads on demand; each call to `iterator()` returns an independent `LoadIterator`. 1213
Listing 13.6 `load_plan.py`: an iterable with a hand-written iterator class, and the same iterable written with a generator function. 1215
Listing 13.7 `ConcurrentModification.java`: `set` during iteration is allowed; `add` and `remove` through the list throw `ConcurrentModificationException`; one case slips through silently; three correct alternatives. 1219
Listing 13.8 `concurrent_modification.py`: a Python list changed during a `for` loop skips elements silently; dictionaries and sets raise `RuntimeError`; three correct alternatives. 1222
Listing 13.9 `AgeSort.java`: a hand-written `AgeComparator` that also handles `null`, and the same orders built with `Comparator.comparingInt`, `reversed` and `nullsFirst`. 1225
Listing 13.10 `age_sort.py`: sorting with a key function (`attrgetter`, `lambda`), `reverse=True`, and a Java-style comparator adapted with `functools.cmp_to_key`. 1228
Listing 13.11 `NameOrder.java`: a `compareTo` that handles only −1, 0 and 1, a correct natural order, a last-then-first comparator, and why comparing glued strings is fragile. 1229
Listing 13.12 `name_order.py`: `@dataclass(order=True)` for a field-by-field natural order, tuple keys for other orders, and the broken comparison reproduced with `cmp_to_key`. 1232
Listing 13.13 `CopyDemo.java`: the same rename-and-add experiment on an alias, a shallow copy and a deep copy of a list of `Person` objects. 1235
Listing 13.14 `copy_demo.py`: aliasing, `list(...)` as a shallow copy, `copy.deepcopy`, and the same three levels for a nested list. 1238
Listing 13.15 `MedianDemo.java`: a median method that sorts its caller's list, one that sorts a copy, and an unmodifiable view that turns the hidden side effect into an exception. 1240
Listing 13.16 `median_demo.py`: `list.sort()` changes the caller's list, `sorted()` does not, and a tuple cannot be sorted in place at all. 1241
Listing 13.17 `ImmutableCollections.java`: `final` versus immutable, an unmodifiable view that shows later changes, an unmodifiable copy that does not, the `of` factory methods, and shallow unmodifiability. 1243
Listing 13.18 `immutable_collections.py`: `Final` as a hint, `MappingProxyType` as a read-only view, `tuple` and `frozenset` as immutable copies, hashability, and shallow immutability. 1245
Listing 13.19 `ImmutableClasses.java`: an immutable `Point` written by hand, a `Student` with a life-long id, a class that leaks its list, and a record with a defensive copy in its compact constructor. 1248
Listing 13.20 `immutable_classes.py`: frozen dataclasses with `replace`, an id fixed at birth, a leaky class, a defensive copy in `__post_init__`, and `NamedTuple`. 1251
Listing 13.21 `MutableKey.java`: a mutable `Spot` used as a key in a `HashMap` and a `HashSet`, changed while stored; then an immutable record used correctly. 1254
Listing 13.22 `mutable_key.py`: the same lost key in a `dict` and a `set`, the frozen-dataclass cure, and Python's refusal to hash a list. 1257
Listing 13.23 `StreamsDemo.java`: a loop and a stream computing the same total, laziness made visible with `peek`, single use, collectors, and an infinite stream limited with `limit`. 1260
Listing 13.24 `streams_demo.py`: comprehensions and generator expressions, `map`, `filter` and `reduce`, laziness, single use, grouping, and infinite iterators from `itertools` 1262
Listing 13.25 `LaundryLedger.java`: Profitina Laundry with an immutable `Money` record, price rules, frozen orders, an append-only `OrderHistory` with two custom iterators, and a report built from streams. 1265
Listing 13.26 `laundry_ledger.py`: the same ledger with frozen dataclasses, a generator-based newest-first iterator, a lazy per-customer generator, and a report built from comprehensions and `itertools.groupby` 1273
Listing 13.27 `PickupDays.java`: an `Iterable<LocalDate>` that skips Sundays, with an anonymous inner iterator. 1283
Listing 13.28 `pickup_days.py`: the same iterable as a generator. 1285
Listing 13.29 `LedgerQueries.java`: five further questions answered with streams, including `partitioningBy`, `flatMap`, `max`, `counting` and `flatMapping`. 1287

Listing 13.30 `ledger_queries.py`: the same five questions with comprehensions, `max` with a key, `Counter` and `next` with a default. 1289
Listing 13.31 `MoneySplit.java`: splitting and allocating `Money` so that the parts always add up to the total. 1290
Listing 13.32 `money_split.py`: the same with `divmod` and a tuple of `Money` values. 1291
Listing 14.1 `ParentChild.java`: a parent starts two counting children as threads, sleeps for three seconds, then interrupts them and waits for them to end. 1297
Listing 14.2 `parent_child.py`: the same parent and children in Python; a shared `threading.Event` replaces Java's interrupt. 1300
Listing 14.3 `StopFlag.java`: a thread spinning on a plain `boolean` flag may never see the flag change; the same loop on a `volatile` flag stops at once. 1302
Listing 14.4 `stop_flag.py`: in CPython a plain attribute used as a stop flag does become visible; a `threading.Event` is still the idiomatic choice. 1304
Listing 14.5 `CreateThreads.java`: a subclass of `Thread`, a `Runnable`, a lambda, and an `ExecutorService` that returns results through `Future` objects; and what happens if you call `run()` instead of `start()`. 1305
Listing 14.6 `create_threads.py`: a subclass of `threading.Thread`, a target function, a lambda, and a `ThreadPoolExecutor` with `submit` and `map`. 1307
Listing 14.7 `ThreadStates.java`: one thread in each of the six states of `Thread.State`, observed with `getState()`. 1311
Listing 14.8 `thread_states.py`: what Python lets us observe about a thread's life: `is_alive()`, `ident`, `Lock.locked()` and `threading.enumerate()`. 1312
Listing 14.9 `SpeedupDemo.java`: counting primes below three million in one thread and in two threads, with timings. 1314
Listing 14.10 `gil_demo.py`: CPU-bound work in one thread, two threads and two processes, and waiting in one thread and four threads. 1315
Listing 14.11 `RaceDemo.java`: three children share one counter without any lock; five rounds of increment-and-decrement and five rounds of increment only. 1318
Listing 14.12 `race_demo.py`: lost updates on a shared counter in Python, with a plain `+=` and with a deliberately yielding read-modify-write. 1323
Listing 14.13 `SafeCounters.java`: five correct counters: synchronized methods, a synchronized block in the caller, a `ReentrantLock`, an `AtomicInteger` and a `LongAdder`. 1325
Listing 14.14 `safe_counters.py`: the slow read-modify-write counter made correct with `threading.Lock`, written with `with` and with explicit `acquire`/`release`. 1329
Listing 14.15 `ProCon.java`: a producer and a slower consumer share a five-cell circular buffer; `wait()` and `notifyAll()` make each wait for the other. 1332
Listing 14.16 `procon.py`: the same circular buffer with `threading.Condition`. 1336
Listing 14.17 `QueueDemo.java`: the producer–consumer exchange with an `ArrayBlockingQueue` of capacity five and a poison pill. 1339
Listing 14.18 `queue_demo.py`: the same exchange with `queue.Queue(maxsize=5)` and `None` as the poison pill. 1341
Listing 14.19 `Philosopher.java`: five philosophers, each a thread, each taking the left fork and then the right fork; `main` draws the table until every philosopher holds exactly one fork. 1342
Listing 14.20 `philosopher.py`: the dining philosophers in Python; each fork is a `threading.Lock`. 1346
Listing 14.21 `DeadlockDemo.java`: two threads take two locks in opposite orders; after one second the JVM's deadlock detector reports the cycle. 1348
Listing 14.22 `deadlock_demo.py`: the two clerks in Python; after one second both threads are still alive and both locks are still held. 1350
Listing 14.23 `DeadlockFixes.java`: the left-then-right philosophers deadlock; lock ordering, `tryLock` with a timeout, and a four-seat semaphore all finish their fifteen meals. 1351
Listing 14.24 `deadlock_fixes.py`: the naive philosophers and three cures in Python. 1354
Listing 14.25 `Livelock.java`: two workers who always give way hand a resource back and forth without ever using it; limiting their politeness to one handover ends the livelock. 1357
Listing 14.26 `livelock.py`: the two over-polite workers in Python. 1361

Listing 14.27 `LaundryShop.java`: three washing-machine threads take orders from a `LinkedBlockingQueue` and pay into a synchronized `Ledger`; poison pills switch them off. 1364
Listing 14.28 `laundry_shop.py`: the concurrent laundry in Python: `WashingMachine` subclasses `threading.Thread`, the ledger guards its totals with a `Lock`, and `None` is the poison pill. 1369
Listing 14.29 `WalletTransfer.java`: transfers in both directions between two wallets, made deadlock-free by locking the smaller id first. 1377
Listing 14.30 `wallet_transfer.py`: the same deadlock-free transfers in Python. 1379
Listing 14.31 `ExpressQueue.java`: a `PriorityBlockingQueue` lets express orders jump ahead of normal ones. 1380
Listing 14.32 `express_queue.py`: express orders first with `queue.PriorityQueue` and tuples. 1381
Listing 15.1 `UdpHello.java`: three UDP datagrams sent over loopback from one socket to another. 1388
Listing 15.2 `udp_hello.py`: the same three datagrams in Python. 1390
Listing 15.3 `EchoServer.java`: accept one client and send every line back to it. 1394
Listing 15.4 `echo_server.py`: the echo server in Python. 1395
Listing 15.5 `EchoClient.java`: send lines typed by the user and print what comes back. 1396
Listing 15.6 `echo_client.py`: the echo client in Python. 1397
Listing 15.7 `KnockKnockProtocol.java`: the protocol of **Table 15.3** as a state machine, with a `main` that plays a scripted conversation. 1402
Listing 15.8 `knock_knock_protocol.py`: the same state machine in Python. 1404
Listing 15.9 `KnockKnockServer.java`: a knock-knock server that talks to many clients at once, one pool thread per client, with a clean `stop()` 1406
Listing 15.10 `knock_knock_server.py`: the multi-client server in Python. 1410
Listing 15.11 `KnockKnockClient.java`: talk to the knock-knock server from the keyboard. 1412
Listing 15.12 `knock_knock_client.py`: the knock-knock client in Python. 1413
Listing 15.13 `KnockKnockLoadTest.java`: start a server, run three clients at the same time against it, and stop it. 1415
Listing 15.14 `knock_knock_load_test.py`: the same concurrency test in Python. 1418
Listing 15.15 `Json.java`: a minimal JSON reader and writer; objects become `LinkedHashMap`, arrays `ArrayList`, whole numbers `Long`. 1421
Listing 15.16 `json_roundtrip.py`: the same round trips with Python's standard `json` module. 1428
Listing 15.17 `LaundryStatusServer.java`: the Profitina Laundry order-status service, with its domain classes `Status`, `LaundryOrder` and `OrderBook`. 1429
Listing 15.18 `laundry_status_server.py`: the order-status service in Python. 1437
Listing 15.19 `LaundryStatusClient.java`: a client object that hides the wire format, and a scenario that exercises every rule of **Table 15.4**. 1444
Listing 15.20 `laundry_status_client.py`: the client object in Python. 1448
Listing 15.21 `InteropDemo.java`: start the Python server as a separate process and talk to it with the Java client. 1451
Listing 15.22 `interop_demo.py`: start the Java server as a separate process and talk to it from Python. 1453
Listing 15.23 `HttpStatusDemo.java`: an HTTP server on localhost, queried with `HttpClient` and with a bare socket. 1456
Listing 15.24 `http_status_demo.py`: the same server and questions in Python. 1459
Listing 15.25 `LaundryStatusCheck.java`: ten automated checks, each against its own server on an ephemeral port. 1462
Listing 15.26 `test_laundry_status.py`: the same ten tests with `unittest` 1466
Listing 15.27 `RetryDemo.java`: connecting with exponential backoff; the server appears after the third failure. 1481
Listing 15.28 `retry_demo.py`: the same retrying client in Python. 1483
Listing 15.29 `ChatRoom.java`: a chat room that broadcasts every line, with scripted members for a repeatable demonstration. 1486
Listing 15.30 `chat_room.py`: the chat room in Python. 1491

Listing 16.1 `JokeStore.java`: the storage contract, with the `Joke` record and an in-memory stub built by `JokeStore.of`. 1504
Listing 16.2 `JokeStores.java`: a file-backed `JokeStore`, a one-based store that breaks the contract, and one contract test run against all three stores. 1505
Listing 16.3 `joke_store.py`: the storage contract as an abstract base class, the frozen `Joke` dataclass and the in-memory stub `ListJokeStore`. 1508
Listing 16.4 `joke_stores.py`: the file store, the one-based store and the contract, with `unittest` tests. 1509
Listing 16.5 `KnockKnockProtocol.java`: the transition table as a class, a stateless wrong version, and a `play` helper that runs a conversation without a network. 1514
Listing 16.6 `knock_knock_protocol.py`: the same state machine in Python, with the same tests. 1517
Listing 16.7 `KnockKnockServer.java`: an acceptor thread, one worker per client, a static `converse` method that runs one conversation, and a deliberately wrong one-at-a-time mode. 1521
Listing 16.8 `KnockKnockClient.java`: the client loop, the scripted `PoliteUser`, an interactive mode and a self-test that also shows the one-at-a-time server starving a second client. 1524
Listing 16.9 `knock_knock_server.py`: the same server with `threading.Thread` workers and a lock around the counters. 1528
Listing 16.10 `knock_knock_client.py`: the client, `PoliteUser` as a callable class, and three tests including the starved second client. 1530
Listing 16.11 `ProtocolContractTest.java`: tests written by each member for the other, with stubs that replay fixed lines, and a mutant client that a weak test lets through. 1535
Listing 16.12 `protocol_contract_test.py`: the cross tests in Python, one test class per member. 1537
Listing 16.13 `ConcurrentClientsTest.java`: twenty clients released together by a start gate, checked against an oracle, and a server that shares one protocol object. 1540
Listing 16.14 `concurrent_clients_test.py`: the same distributed test in Python. 1544
Listing 16.15 `FailureDetection.java`: a client that turns every failure into one sentence, a naive client, deliberately misbehaving servers, and a server that notices a departed client. 1549
Listing 16.16 `failure_detection.py`: the same failure scenarios in Python. 1552
Listing 16.17 `KnockKnockGui.java`: MVC with a `View` interface, a controller that runs the network on its own thread, a Swing view, and a headless self-test. 1556
Listing 16.18 `knock_knock_gui.py`: the same design with Tkinter; the network thread posts view updates into a queue that the Tk thread drains. 1562
Listing 16.19 `PerformanceProbe.java`: latency and throughput of the knock-knock server with 1, 10 and 50 simultaneous clients. 1568
Listing 16.20 `performance_probe.py`: the same measurement in Python. 1570
Listing 16.21 `PickupSchedule.java`: a sorted array with an inner-class iterator, a lazy filtered view and a fail-fast check. 1573
Listing 16.22 `pickup_schedule.py`: the same schedule with an explicit iterator class and a generator for the lazy view. 1578
Listing 16.23 `Money.java`: an immutable value object with validated construction, value equality, a consistent hash code, ordering, and two broken variants. 1583
Listing 16.24 `money.py`: the same value object as a frozen dataclass, and the two broken variants. 1588
Listing 16.25 `WashCounter.java`: an unsafe counter, the same race made certain, and three thread-safe counters, all run by four threads released together. 1592
Listing 16.26 `wash_counter.py`: the same experiment in Python. 1595
Listing 16.27 `WashLine.java`: a hand-made bounded buffer, the same line with `ArrayBlockingQueue`, poison pills, and a buffer that uses `if` instead of `while`. 1599
Listing 16.28 `wash_line.py`: the same wash line in Python. 1604
Listing 16.29 `LaundryProtocol.java`: the protocol as a pure function, a multi-client server, a client with a reply timeout, and a client that forgets to flush. 1609
Listing 16.30 `laundry_protocol.py`: the same protocol, server and clients in Python. 1614
Listing 16.31 `LaundryCore.java`: the model of the capstone: a checked exception, the `PriceRule` interface, the service hierarchy, the status enum, the immutable `Order` record, the file repository, and the thread-safe `Shop`. 1621

Listing 16.32 `ProfitinaLaundry.java`: the controller, the views, the socket API with dot-terminated replies, a client, and the self-test. 1625
Listing 16.33 `laundry_core.py`: the model in Python, with an `RLock` protecting the shop. 1632
Listing 16.34 `profitina_laundry.py`: the controller, the socket API, the client and the tests in Python. 1637
Listing 16.35 `ExerciseSolutions.java`: `Countdown`, correct removal during a loop, `split`, `TOTAL`, and a `READY` counter tested over the network. 1651
Listing 16.36 `exercise_solutions.py`: the same solutions in Python, and the SQLite joke store of Exercise 8. 1655

List of Pseudocode

Pseudocode 1.1 The procedure for taking a laundry order, written precisely enough for someone who has never worked in the shop. 2
Pseudocode 10.1 The event loop that every GUI toolkit runs: wait, take the next event, call the listeners registered for it, and repaint when idle. 875

List of Equations

(1.1) 3
(1.2) 3
(1.3) 26
(1.4) 26
(2.1) 76
(2.2) 92
(2.3) 92
(2.4) 99
(3.1) 151
(3.2) 156
(3.3) 159
(3.4) 160
(3.5) 175
(3.6) 177
(3.7) 177
(3.8) 179
(3.9) 185
(3.10) 185
(3.11) 196
(4.1) 209
(4.2) 209
(4.3) 216
(4.4) 216
(4.5) 216
(4.6) 229
(4.7) 234
(4.8) 286

(4.9) 291
(5.1) 305
(5.2) 318
(5.3) 318
(5.4) 357
(6.1) 393
(6.2) 405
(6.3) 414
(6.4) 432
(6.5) 447
(6.6) 465
(6.7) 468
(6.8) 468
(6.9) 469
(7.1) 546
(7.2) 554
(7.3) 554
(7.4) 570
(7.5) 575
(7.6) 581
(7.7) 586
(7.8) 586
(7.9) 586
(7.10)....586
(8.1) 622
(8.2) 652
(8.3) 657
(8.4) 668
(8.5) 683
(8.6) 691
(8.7) 701
(9.1) 755
(9.2) 755
(9.3) 767
(9.4) 776
(9.5) 776
(9.6) 777
(9.7) 809
(9.8) 834
(9.9) 834
(10.1)....901
(10.2)....905
(10.3)....906
(10.4)....923
(10.5)....932
(10.6)....932
(10.7)....932
(10.8)....961
(11.1)....974
(11.2)....988
(11.3)....999

(11.4) 1022
(11.5) 1022
(11.6) 1022
(12.1) 1076
(12.2) 1079
(12.3) 1088
(12.4) 1093
(12.5) 1093
(12.6) 1093
(12.7) 1105
(12.8) 1113
(12.9) 1121
(12.10) 1167
(13.1) 1204
(13.2) 1219
(13.3) 1224
(13.4) 1224
(13.5) 1225
(13.6) 1254
(13.7) 1254
(13.8) 1259
(13.9) 1265
(13.10) 1290
(14.1) 1315
(14.2) 1322
(14.3) 1322
(14.4) 1331
(14.5) 1351
(14.6) 1369
(15.1) 1385
(15.2) 1386
(15.3) 1406
(15.4) 1455
(15.5) 1455
(15.6) 1461
(16.1) 1514
(16.2) 1568
(16.3) 1568
(16.4) 1582
(16.5) 1591
(16.6) 1591
(16.7) 1658

Preface

Every useful program you have ever used — the app that tells you when your laundry is ready, the game on your phone, the website where you pay a bill — is built out of *objects*: small, self-contained pieces of software that each know something and can do something, and that cooperate by sending one another messages. This book teaches you to think in objects and to build with them, in the two languages that the world uses most for exactly that purpose: **Java** and **Python**.

It grew out of the Object-Oriented Programming course (EF234302) taught in the Department of Informatics at Institut Teknologi Sepuluh Nopember (ITS), Surabaya, and it has been rebuilt so that it stands entirely on its own. Its sixteen chapters follow the rhythm of a sixteen-week semester, but you need no classroom, no lecturer and no other document to learn from it.

Who this book is for

You are assumed to know nothing. If you have never written a program, never installed a programming tool, never heard the word *class* used outside a school building, you are exactly the reader this book was written for. Chapter 1 starts with an empty computer and ends with your first two programs running, one in each language. From there, every idea is introduced in plain words, then shown in Java, then shown in Python, then run, and then explained line by line — because a program you have run and understood is worth ten you have only read.

Why two languages

Java and Python are both object-oriented, but they make different choices: Java checks types before the program runs and asks you to say everything explicitly; Python checks at run time and lets you say much less. Seeing the same idea written both ways is the quickest route to understanding the idea itself rather than the habits of one language. Every example in this book is therefore a pair — a Java program and a Python program that do the same thing and print the same output — and both are complete: you can copy either one into a file and run it.

What is in it

Chapters 1 to 3 set up your workbench (VS Code, the Java Development Kit and Python, on Windows, macOS or Linux), teach the development environment and the debugger, and build the first classes: objects, strings, types, conditionals and loops. Chapters 5 to 7 add arrays and lists, input, exceptions and files, documentation and testing, interfaces, inheritance, polymorphism and access control. Chapters 9 to 11 cover collections and generics, graphical user interfaces and events, and the Model–View–Controller pattern. Chapters 13 to 15 treat iterators and immutability, threads and their dangers, and networking with sockets, ending with a team project.

Chapters 4, 8, 12 and 16 are **Evaluation and Discussion** chapters. Each closes one quarter of the book by solving complete problems in full — every problem restated, designed on paper, programmed in both languages, tested, run, and explained — with the tempting wrong solutions shown and taken apart, and a “check yourself” variant at the end of every problem.

Throughout, one small business grows chapter by chapter: **Profitina Laundry**, a laundry shop whose customers, orders, services, machines and receipts become classes, interfaces, collections, windows, threads and network services as the book goes on. Each chapter also has a *Profitina in Practice* section that shows where the same ideas appear in a real product.

Every program in this book was compiled and run before its output was printed. The programs are collected, chapter by chapter, in the accompanying `Code` folder, with a README in every chapter folder that explains how to run each one in a terminal and in VS Code on Windows, macOS and Linux.

How to read it

With the computer on. Type the programs in rather than only reading them; run them; change one thing and run them again; break them on purpose and read the error message until it makes sense. The exercises at the end of each chapter all have complete solutions — use them after your own attempt, not instead of it.

Numbered figures, tables, listings, pseudocode and equations are all live links: in the electronic edition, click "Listing 5.3" or "[4]" anywhere in the text and you are taken straight to it. The contents and the lists at the front of the book work the same way.

About the author

Irfan Subakti describes himself in one line — *Father | Educator | Entrepreneur | Bridging Tech & Nature: CS Lecturer × Crypto/Digital Business × AI/ML Specialist × Fruit Farmer | Storyteller* — and every part of that line has found its way into this book. He has taught in the Department of Informatics at ITS since 1998. He supervises and examines undergraduate theses, teaches MLOps for the IBM SkillsBuild program, and builds products: he founded Profitina, an AI-visibility platform whose engineering appears in the "Profitina in Practice" section of every chapter. Away from the keyboard he tends the family fruit and durian orchard, where he learned that good software, like a good orchard, is grown patiently, one well-tended piece at a time. The storyteller in him is the reason this book prefers a story to a list whenever a story will do.

Acknowledgments

This book owes its shape to the students who worked through its earlier forms and asked the questions that showed where an explanation was thin, and to the colleagues at ITS with whom the course has been taught and refined over the years. It owes its standard to the classic texts on object-oriented design and to the official Java and Python documentation, which are cited wherever they are used.

Surabaya, September 2026



Irfan Subakti · yifana@gmail.com

Profitina

profitina.com

Chapter 1

Objects, Programs, and Your First Workbench

By the end of this chapter you will know what a program is and how a computer comes to run one; you will be able to look at an ordinary place, such as a small laundry shop, and see in it the *objects* that an object-oriented program would be built from, each with its own *state* and its own *behavior*, each answering the *messages* that other objects send it; you will know why this book teaches in two languages, Java and Python, and what each of them is good at; you will have a working programmer's workbench on your own computer, whether it runs Windows, macOS or Linux, with the Java Development Kit, the Python interpreter and Visual Studio Code installed, checked and connected; and you will have typed, run and debugged your first programs in both languages, ending with your first real class, a laundry `Order` that knows its own price and refuses to be collected before it is clean.

1.1 Where We Begin

On a street corner in Surabaya there is a small laundry shop. It has a counter with a scale on it, three washing machines along the back wall, two dryers, an ironing board, a shelf of plastic bags tied with colored string, and a thick notebook. The owner, Bu Sari, opens at seven in the morning. A customer walks in with a bag of clothes. Bu Sari puts the bag on the scale — three and a half kilograms — asks whether the customer wants the ordinary service or the express one, writes a line in the notebook, tears off a ticket, and names a price. The clothes go into a machine, then into a dryer, then onto the folding table, then into a bag on the shelf. Two days later the customer comes back with the ticket, pays, and carries the clean clothes home.

Nothing in that story mentions a computer, and yet it is already full of the ideas this book is about. There is a **procedure** — weigh, ask, write, price, wash, dry, fold, hand over — that is followed the same way every time. There is **information** — the customer's name, the weight, the kind of service, the price, whether the clothes are still in the machine or already on the shelf. And there are **things**, each with its own information and its own abilities: the ticket knows its number and its price; a washing machine knows whether it is running and how much it can hold; the notebook knows every order of the day. When Bu Sari hires a second worker, she does not explain the whole shop in one breath. She says: *this is how a ticket works, this is how a machine works, this is what you say to a customer*. She explains the shop one kind of thing at a time.

A computer program for this shop can be written the same two ways that Bu Sari could explain it. It can be written as one long list of instructions that shuffles numbers around, or it can be written as a small society of **objects** — tickets, machines, customers — each of which looks after its own information and does its own jobs when asked. The second way is called **object-oriented programming**, and it is how most of the software you use every day is organized: the banking application on your phone, the web shop where you buy a charger, the game you play on the bus, the editor you will install later in this chapter.

This book teaches object-oriented programming from the very beginning, in two languages side by side: **Java** and **Python**. Every idea is first explained in plain words, usually with the laundry shop as the example. Then it is shown as a complete Java program, then as a complete Python program that does exactly the same thing, then both are run and their real output is printed, and then every line is explained. You will never be asked to accept a program you cannot run yourself.

The book assumes nothing. If you have written C or C++ and know words like *variable*, *function*, *array* and *pointer*, you will move quickly through the early pages, and the book will point out where Java and Python agree with C and where they part company. If you have never written a line of code or opened a terminal window, you are just as welcome: every term is explained when it first appears, every installation step is written out for Windows, macOS and Linux, and every program is complete and prints exactly what the book says, because the output you read was produced by running it.

The chapter walks from the shop counter to the keyboard: what a program is (Section 1.2), how to see objects (Sections 1.3 and 1.4), why Java and Python (Section 1.5), the tools and their installation (Sections 1.6 and 1.7), your first class (Section 1.8), the road ahead (Section 1.9), and these ideas in a real product (Section 1.10).

1.2 What a Program Is

1.2.1 A recipe for the counter

Suppose Bu Sari goes on holiday and leaves her nephew Dimas in charge. Dimas is willing but has never worked in a laundry shop. So she writes down, on a card taped to the counter, exactly what to do when a customer arrives. **Pseudocode 1.1** is that card, written a little more carefully than a person usually would.

Pseudocode 1.1 The procedure for taking a laundry order, written precisely enough for someone who has never worked in the shop.

```
TAKE-ORDER(customer, laundry)
1  w = weight of the laundry on the scale, in kilograms
2  ask the customer: regular or express?
3  p = round(7000 * w)                      // Rp 7,000 per kilogram
4  if the customer chose express
5      p = p + floor(p / 2)                 // express costs half as much again
6  write a ticket: number, customer, w, express or not, p, status "RECEIVED"
7  give the customer the ticket stub and ask for p rupiah
```

Look at what makes the card useful. Each line says one thing. Nothing is left to taste: “round” and “floor” say exactly what to do with fractions of a rupiah. The lines are followed in order, top to bottom, except that line 5 is done only *if* the condition on line 4 is true. The card mentions two pieces of information that change from customer to customer — the weight w and the price p — and gives each a short name so that later lines can refer to it. A card like this, a finite list of precise steps that turns some input (a customer and a bag of laundry) into some output (a ticket and a price), is called an **algorithm**.

> **DEFINITION.** An *algorithm* is a finite sequence of precise steps that transforms an input into an output. A *program* is an algorithm written in a programming language, so exactly that a computer can carry it out.

The prices on the card can be written as two small formulas. For a weight of w kilograms, the regular price in rupiah is

$$p(w) = \text{round}(7000\, w) \tag{1.1}$$

and the express price adds half of that, rounded down to a whole rupiah:

$$p_{\text{express}}(w) = p(w) + \left\lfloor \frac{p(w)}{2} \right\rfloor \tag{1.2}$$

The symbol $\lfloor x \rfloor$, read "floor of x", means the largest whole number that is not bigger than x; so $\lfloor 12250.0 \rfloor = 12250$ and $\lfloor 7.9 \rfloor = 7$. Let us try Equation (1.1) and Equation (1.2) on Ani's bag of 3.5 kilograms: $p(3.5) = \text{round}(7000 \times 3.5) = \text{round}(24500) = 24500$, so a regular wash costs Rp 24,500. On Budi's 5 kilograms, express: $p(5) = 35000$, and $p_{\text{express}}(5) = 35000 + \lfloor 35000/2 \rfloor = 35000 + 17500 = 52500$. Keep those two numbers in mind; you will see the computer print them several times in this chapter.

1.2.2 Languages for people, languages for machines

Dimas can read Bu Sari's card because it is written in Indonesian, or English, or whatever language they share. A computer cannot read either. The processor inside a computer understands only its own **machine language**: long sequences of numbers, each of which means something tiny, such as "add these two numbers" or "copy this number to that place in memory". Writing a laundry program directly in machine language would be like writing Bu Sari's card one muscle movement at a time.

A **programming language** sits between the two. It is precise enough that a computer program can translate it into machine language without guessing, and readable enough that a person can write it, read it next week, and explain it to a colleague. Java and Python are two such languages. Here is line 3 of the card in each of them; do not worry yet about every symbol:

```
long p = Math.round(7000 * w);      Java
p = round(7000 * w)                 Python
```

The two lines say the same thing in slightly different grammar. Java insists that you say what *kind* of value `p` will hold (`long`, a whole number that can be very large) and ends every statement with a semicolon. Python lets the value decide its own kind and ends a statement at the end of the line. Chapter 3 explores these differences carefully; for now it is enough to see that both are far closer to the card than to machine language.

The text you write in a programming language is called **source code**, and a file that holds it is a **source file**. Java source files end in `.java`; Python source files end in `.py`. They are plain text: you could write one in Notepad, although in Section 1.7 you will install something much more comfortable.

1.2.3 From source code to a running program

Somebody has to translate your source code into instructions the processor can follow. There are two classic ways of doing it, and Java and Python each use a little of both. **Figure 1.1** shows the two journeys side by side.

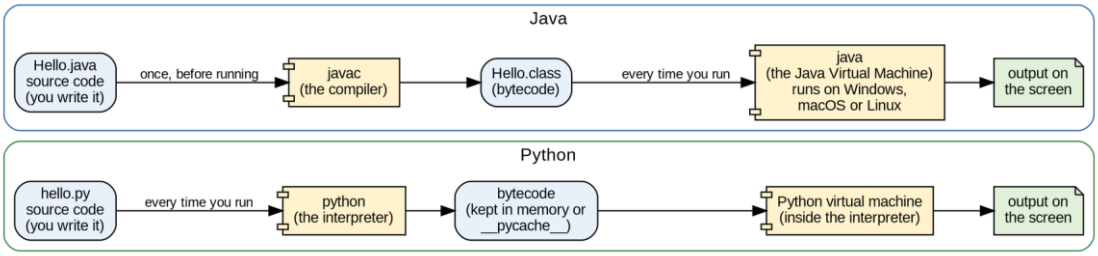


Figure 1.1 How source code becomes a running program: Java compiles once to bytecode that the Java Virtual Machine runs; Python compiles to bytecode each time it runs, inside the interpreter.

A **compiler** translates a whole source file in advance, producing a new file that can be run later, many times, without the source. An **interpreter** reads the source and carries it out straight away, each time you run it. A compiler is like a translator who translates a whole book once and prints it; an interpreter is like a translator standing next to you, translating aloud as you go.

Java uses a compiler called `javac`. It reads `Hello.java` and writes `Hello.class`. But `Hello.class` does not contain the machine language of your particular processor. It contains **bytecode**: instructions for an imaginary, standardized computer called the **Java Virtual Machine** (JVM). The command `java` starts a JVM, which reads the bytecode and runs it on the real processor underneath. Because every JVM understands the same bytecode, the very same `Hello.class` runs on Windows, on macOS, on Linux, on a server in a data center and, in adapted form, on phones and smart cards. This is the famous promise of Java, *write once, run anywhere*, and Section 1.5 returns to it.

Python looks, from the outside, like a pure interpreter: you type `python hello.py` and the program runs. Inside, the interpreter first compiles your source into Python bytecode — very quickly, and usually without saving it, although for files that other files import it keeps a copy in a folder called `__pycache__` — and then a Python virtual machine inside the interpreter carries out that bytecode. You never run a separate compiler, which is part of why Python feels so light to start with.

The practical difference is simple. Java has two steps, **compile** (`javac`) then **run** (`java`), and after changing the source you must compile again or you will run the old version. Python has one step: **run** (`python`). Section 1.7 shows how VS Code hides even these steps behind a single button.

1.2.4 Data and the operations on it

Every program, however large, is made of two kinds of stuff. There is **data**: the numbers, words and yes-or-no facts the program works with — 3.5 kilograms, the name “Ani”, the fact that an order is express. And there are **operations**: the things done to the data — multiplying the weight by the price per kilogram, comparing a status with “READY”, printing a line. In **Pseudocode 1.1**, lines 1 and 6 mostly gather and record data, and lines 3 and 5 mostly operate on it.

Much of the history of programming languages is a story about keeping data and operations *organized* as programs grow from ten lines to ten million. The early answer was to keep them apart: data in one place, operations as separate procedures that receive the data. The object-oriented answer, introduced next, is to put each piece of data together with the operations that belong to it. For programs that model a world full of things — shops, banks, games, hospitals — the second answer has proved easier to understand, easier to change and much harder to break by accident.

1.3 Thinking in Objects

1.3.1 Looking around the shop

Stand at Bu Sari's counter again and look around, this time as a programmer. Ask of every *thing* you see two questions: *what does it know?* and *what can it do?*

The **order ticket** knows its number, the customer's name, the weight, whether the service is express, and where the laundry is right now: just received, in the washing machine, ready on the shelf, or already collected. It can tell you its price. It can be moved along: from received to washing, from washing to ready, from ready to picked up.

A **washing machine** knows its name ("Machine A"), how many kilograms it can hold, how many kilograms are inside right now, and whether it is running. It can be loaded, it can be started, and it can finish a cycle and be emptied. And it has opinions: it refuses a load that would make it too heavy, and it refuses to be loaded while it is running.

A **customer** knows his or her name and telephone number, and can hand over laundry, pay, and collect.

The **cashier** — Bu Sari, or Dimas — knows the price list and the day's takings, and can take an order, receive payment and give change.

Table 1.1 collects these observations. Read it slowly: this table, and not any code, is where an object-oriented program begins.

Table 1.1 The things in a laundry shop, each described by what it knows and what it can do.

Thing in the shop	What it knows (state)	What it can do (behavior)
Order ticket	number, customer, weight, express or not, status	tell its price; start washing; become ready; be picked up
Washing machine	name, capacity, kilograms inside, running or not	accept a load; start; finish and be emptied
Customer	name, telephone number	hand over laundry; pay; collect
Cashier	price list, today's cash	take an order; receive payment; give change

1.3.2 State, behavior and identity

The two columns of **Table 1.1** have names. What an object knows is its **state**. What it can do is its **behavior**. A third property is so obvious in the real world that we rarely notice it: every object is *itself* and no other. Two customers may bring in bags of exactly the same weight on the same morning for the same service, and their tickets may carry exactly the same weight, service and price — but they are still two tickets, and handing Ani's clean clothes to Budi would be a serious mistake. This property is called **identity**.

> **DEFINITION.** An *object* is a thing in a program that has *state* (the information it holds at a given moment), *behavior* (the operations it can perform when asked) and *identity* (it is distinguishable from every other object, even one with identical state).

State changes over time; that is what makes an object interesting. At seven in the morning Ani's ticket says "RECEIVED"; at ten it says "WASHING"; at two in the afternoon it says "READY". The ticket is the same ticket throughout — its identity does not change — but its state does. Behavior is *how* state changes: the ticket does not jump from "RECEIVED" to "PICKED UP" by magic, but

through a sequence of well-defined steps, and some steps are refused at some moments. You cannot pick up laundry that is still in the machine.

1.3.3 Classes: the blank form

Bu Sari does not design a new ticket for every customer. She has a pad of **blank forms**, all printed the same: a space for the number, a space for the name, a space for the weight, a box to tick for express, a space for the price. Each morning she fills in one form per customer. The blank form describes what every ticket will know; each filled-in ticket is a particular ticket with particular values.

In object-oriented programming the blank form is called a **class**, and each filled-in ticket is an **object**, also called an **instance** of the class. The class `Order` says: *every order has a number, a customer, a weight, an express flag and a status; and every order can tell its price, start washing, become ready and be picked up*. The objects `ani` and `budi` are two particular orders, each with its own values. **Figure 1.2** draws the class and two of its objects in the notation programmers use, the **Unified Modeling Language** (UML) [1].

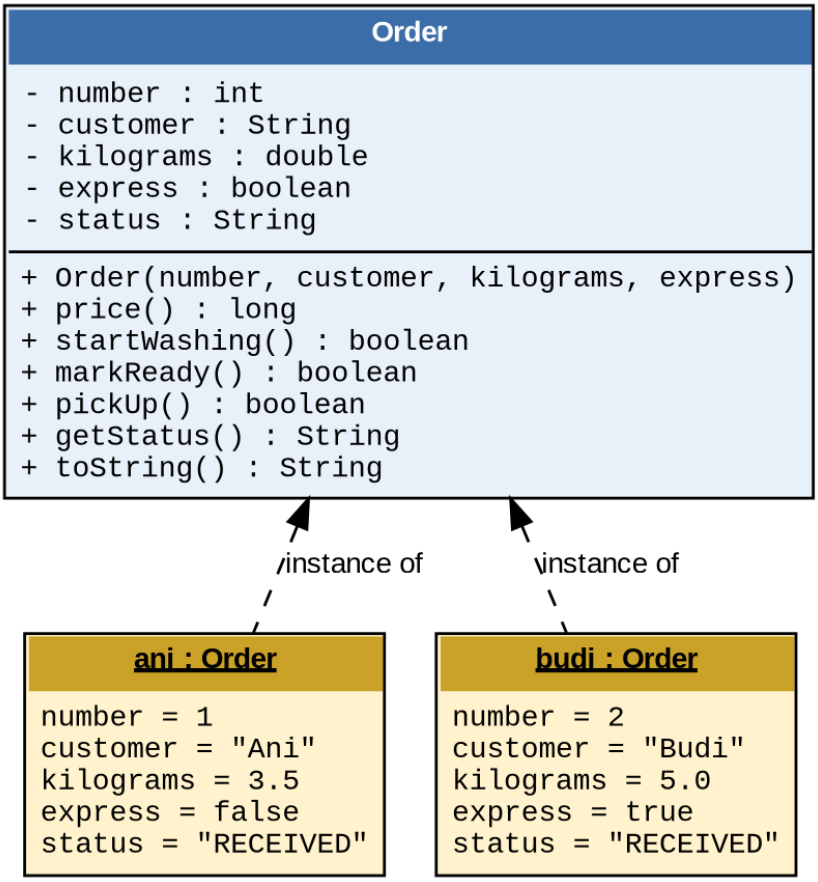


Figure 1.2 The class `Order` (top) is the blank form: it lists the state every order holds and the behavior every order offers; the objects `ani` and `budi` (bottom) are two filled-in tickets.

Read the class box from top to bottom. The first compartment gives the class's name. The second lists its **fields** — the pieces of state every object will carry — each with its type: `int` for a whole number, `String` for text, `double` for a number with a fractional part, `boolean` for true-or-false. The minus sign in front of each field means *private*: only the order itself may look at or change it. The third compartment lists the **methods** — the behavior — with a plus sign meaning *public*: anyone

may ask the order to do these things. The first method, `Order(number, customer, kilograms, express)`, is special: it is the **constructor**, the operation that fills in a new blank form.

The object boxes are drawn differently on purpose. Their names are underlined and written as `name : Class`, and they show *values* instead of types. Both objects have a field called `customer`, but in one it holds `"Ani"` and in the other `"Budi"`. Each object has its own copy of every field; changing Ani's status leaves Budi's alone.

> **KEY INSIGHT.** A class is written once, by a programmer, as source code. Objects are created while the program runs, as many as needed — one per customer, one per machine — and each carries its own state. When people say "write a class", they mean "design the blank form"; when they say "create an object", they mean "fill one in".

1.3.4 Messages: how objects work together

A shop is not a pile of things; it is things *working together*. The customer hands laundry to the cashier. The cashier fills in a ticket and asks it for the price. The ticket works the price out and answers. The cashier tells the customer, then puts the clothes into a machine and presses start. In an object-oriented program, every one of these interactions is a **message**: one object asks another to perform one of its behaviors, sometimes handing over some information with the request, and sometimes receiving an answer back. **Figure 1.3** numbers the messages of one ordinary morning.

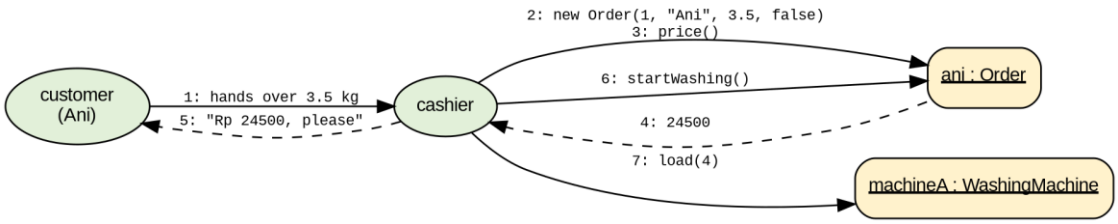


Figure 1.3 One morning at the counter as a conversation between objects: solid arrows are messages, dashed arrows are answers, and the numbers give the order in which they happen.

Follow the numbers. In message 1 the customer hands over 3.5 kilograms. In message 2 the cashier creates a new order — in Java that is literally written `new Order(1, "Ani", 3.5, false)`. In message 3 the cashier asks the order for its price, and in answer 4 the order replies `24500`, which it computed with Equation (1.1) from its own weight. The cashier passes that on to the customer (answer 5), then tells the order that washing has started (message 6) and loads four kilograms into machine A (message 7).

Two things deserve attention. First, the cashier *does not compute the price*: it asks the order, which computes it from information only the order holds, so if the price list changes next year only the order's price method changes. Second, the message names are the names of methods in the class: `price()`, `startWashing()`, `load(4)`. In Java and Python a message is written with a dot: `ani.price()` means "send the message *price* to the object `ani`". The word *message* comes from the early object-oriented language Smalltalk, whose designers pictured a program as a community of little computers talking to one another [2]; Java and Python programmers more often say "call a method on an object", which means the same thing.

1.3.5 Abstraction: keeping only what matters

A real laundry ticket may carry a coffee stain, a phone number scribbled on the back and a crease down the middle. Our `Order` class has none of these. It keeps only what matters for the job the program must do: take orders, price them, and track them to collection. Deciding what to keep

and what to leave out is called **abstraction**, and it is the first design skill of object-oriented programming. A model that keeps too little cannot do its job (an order without a weight cannot be priced); a model that keeps too much is expensive to build and confusing to use.

Abstraction also works on behavior. The cashier does not need to know *how* an order computes its price — a rounding rule, an express surcharge, next year a members' discount — only that orders answer the message `price()` with a number of rupiah. The *what* is public; the *how* is the order's own business.

1.3.6 The four big ideas, previewed

Everything in the rest of this book grows from the few ideas of this section. Programmers usually summarize object-oriented programming as four big ideas. Here each is given in one paragraph, with the laundry shop as the example and the chapter where it is taught in full.

Encapsulation means bundling each object's state together with the behavior that uses it, and hiding the state behind that behavior, so that the state can change only in ways that make sense. The `Order` of Section 1.8 already does this: its status can move only forward, one step at a time, and only through its own methods. Chapter 3 builds classes properly, and Chapter 7 treats access control in full.

Abstraction, which you have just met, means describing *what* an object can do without committing to *how*. Its sharpest tool is the **interface**: a promise such as "anything that can be washed has a method `wash()`", which a washing machine and a dry-cleaning service may keep in completely different ways. Chapter 6 introduces interfaces.

Inheritance means defining a new class as a special case of an existing one: an `ExpressOrder` is an `Order` with a shorter deadline and a higher price, and it can reuse everything an ordinary order already does. Chapter 7 teaches it.

Polymorphism — from Greek, "many shapes" — means that the same message can produce different, appropriate behavior depending on which object receives it. Ask a regular order and an express order for `price()`, and each answers correctly in its own way, while the cashier sends both the same message and never needs to check which kind it holds. Chapter 7 teaches polymorphism alongside inheritance.

1.4 From Procedures to Objects

1.4.1 The notebook program

To see why programmers moved to objects, let us first write a laundry program the older way, which is called **procedural programming**. It is how Bu Sari's notebook works: a page with columns — customer, kilograms, express, status — and one row per order. In a program, the most direct way to hold such a page is a set of **parallel arrays**: one array per column, where position 0 of every array describes the first order, position 1 the second, and so on. An **array** is a numbered row of boxes of the same kind, and the numbers, called **indices**, start from 0.

Listing 1.1 is that program in Java and **Listing 1.2** in Python. Do not try to understand every symbol yet — Sections 1.7 and 1.8 explain the grammar line by line. Read it for its *shape*: four parallel arrays, a separate function that computes a price, a separate function that prints every row, and then two careless lines.

Listing 1.1 `LaundryLedger.java`: the laundry notebook as four parallel arrays and two stand-alone functions, and two careless lines that damage it.

```java
public class LaundryLedger {
    public static void main(String[] args) {
        // Four parallel arrays: position i of every array describes order i.
        String[] customer = {"Ani", "Budi", "Citra"};
        double[] kilograms = {3.5, 5.0, 2.0};
        boolean[] express = {false, true, false};
        String[] status = {"RECEIVED", "RECEIVED", "RECEIVED"};

        printAll(customer, kilograms, express, status);

        // Two careless lines, written far away in a long program:
        status[2] = "PICKED UP";   // Citra's laundry has not even been washed yet
        kilograms[1] = -5.0;       // a slip of the finger: a minus sign too many

        System.out.println("-- after two careless lines --");
        printAll(customer, kilograms, express, status);
    }

    static long price(double kg, boolean isExpress) {
        long base = Math.round(kg * 7000);          // Rp 7,000 per kilogram
        return isExpress ? base + base / 2 : base;  // express costs 50% more
    }

    static void printAll(String[] customer, double[] kilograms,
                         boolean[] express, String[] status) {
        for (int i = 0; i < customer.length; i++) {
            System.out.println(customer[i] + ": " + kilograms[i] + " kg, Rp "
                    + price(kilograms[i], express[i]) + ", " + status[i]);
        }
    }
}
```

Listing 1.2 `laundry_ledger.py`: the same notebook as four parallel Python lists.

```python
def price(kg, is_express):
    base = round(kg * 7000)                 # Rp 7,000 per kilogram
    return base + base // 2 if is_express else base   # express costs 50% more


def print_all(customer, kilograms, express, status):
    for i in range(len(customer)):
        print(customer[i] + ": " + str(kilograms[i]) + " kg, Rp "
              + str(price(kilograms[i], express[i])) + ", " + status[i])


def main():
    # Four parallel lists: position i of every list describes order i.
    customer = ["Ani", "Budi", "Citra"]
    kilograms = [3.5, 5.0, 2.0]
    express = [False, True, False]
    status = ["RECEIVED", "RECEIVED", "RECEIVED"]

    print_all(customer, kilograms, express, status)

    # Two careless lines, written far away in a long program:
    status[2] = "PICKED UP"   # Citra's laundry has not even been washed yet
    kilograms[1] = -5.0       # a slip of the finger: a minus sign too many

    print("-- after two careless lines --")
    print_all(customer, kilograms, express, status)


main()
```

Both programs print exactly the same thing:

```
Ani: 3.5 kg, Rp 24500, RECEIVED
Budi: 5.0 kg, Rp 52500, RECEIVED
Citra: 2.0 kg, Rp 14000, RECEIVED
-- after two careless lines --
Ani: 3.5 kg, Rp 24500, RECEIVED
Budi: -5.0 kg, Rp -52500, RECEIVED
Citra: 2.0 kg, Rp 14000, PICKED UP
```

1.4.2 What went wrong

The first three lines of output are right, and you can check them with Equation (1.1) and Equation (1.2): $p(3.5) = 24500$ for Ani, $p_{\text{express}}(5) = 52500$ for Budi, $p(2) = 14000$ for Citra. The trouble is in the last three lines. Budi's laundry now weighs *minus* five kilograms and costs *minus* Rp 52,500 — the shop would owe him money. And Citra's order says "PICKED UP", although her clothes have never been washed; if the shop trusted that line, her bag would sit on the shelf forever while the records said she had taken it home.

In **Listing 1.1** the two careless lines sit next to each other under a comment, so they are easy to spot. Now imagine the program grown to twenty thousand lines, written by five people over three years. Any of those lines can write into `status` or `kilograms`, because the arrays are data lying in the open, and no single place in the program is responsible for keeping an order sensible. The rule "status only moves forward, one step at a time" lives only in the programmers' heads — and people forget, leave the company, and make typing mistakes at eleven at night.

There is a second, quieter problem: the four arrays must stay *in step*. Add an order to `customer` but forget `status`, and every row after it is silently misaligned. And when Bu Sari decides to record a telephone number, a fifth array must be added, and every function that receives four arrays must be changed to receive five.

> **WARNING.** "Just be careful" is not a strategy. It works for a two-hundred-line program written by one person in one evening. It fails, reliably, for any program that is shared, changed or kept for more than a few weeks — which is to say, for almost every program that matters.

1.4.3 The object answer

The object-oriented answer is to stop spreading an order across four arrays and to make each order a single **object** that holds its own customer, weight, express flag and status — and that holds, in the same place, the only operations that are allowed to change them. **Figure 1.4** puts the two organizations side by side.

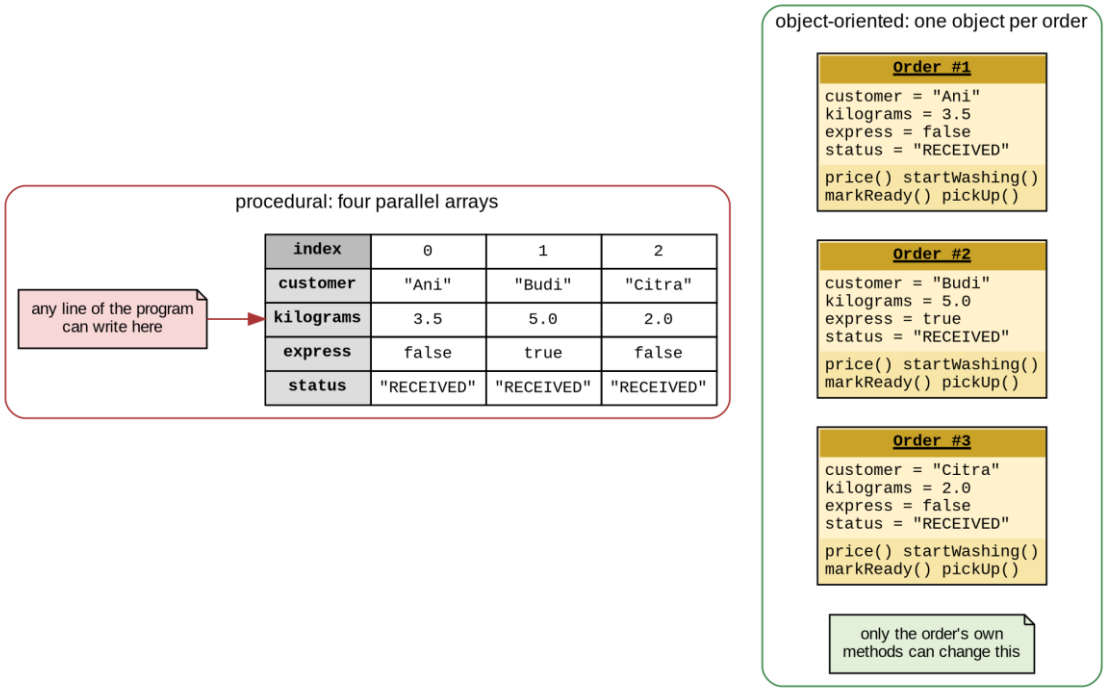


Figure 1.4 The same three orders organized two ways: on the left, four parallel arrays that any line of the program can write to; on the right, one object per order, whose state only its own methods can change.

On the left, every column is open to the whole program. On the right, each order is a closed box with a small set of buttons: `price()`, `startWashing()`, `markReady()`, `pickUp()`. A careless line elsewhere in the program cannot write "PICKED UP" into Citra's order, because the order's status is private; the only way to change it is to press a button, and the `pickUp()` button looks at the current status first and *refuses* if the clothes are not ready. Adding a telephone number means adding one field to one class; no other code needs to change unless it wants to use the new information.

Nothing here lets the object-oriented program compute anything the procedural program cannot. Object orientation is not a new kind of arithmetic. It is a way of *organizing* the same arithmetic so that the program stays understandable and safe as it grows. Section 1.8 builds exactly the right-hand side of **Figure 1.4** in Java and in Python, and you will watch the order refuse a premature pick-up with your own eyes.

1.5 Why Java, and Why Python

There are hundreds of programming languages, and dozens support objects. This book uses two of them, side by side, for reasons that are worth stating plainly.

1.5.1 Java: a short history

Java was created at Sun Microsystems by a team led by James Gosling. It began in the early 1990s as a language for small devices such as television set-top boxes, where a program had to run on many different chips without being rewritten for each. It was released to the public in 1995 as the central part of Sun's *Java platform*, just as the World Wide Web was taking off, and its ability to run the same program on any machine made it an instant fit for the Internet. Sun was acquired by Oracle in 2010, and Oracle has stewarded Java ever since, together with a large open community that develops the reference implementation, **OpenJDK**, in public [3].

Java's grammar was deliberately borrowed from C and C++: curly braces around blocks of code, semicolons at the end of statements, `if`, `for` and `while` written almost exactly as in C. A C programmer can read Java on the first day. What Java left out is just as deliberate. It has fewer *low-level* facilities — the features that work close to the machine's own language, such as pointer arithmetic and manual freeing of memory — because those facilities are exactly where the most dangerous bugs in C and C++ programs come from. Java manages memory itself: when an object is no longer used, a part of the JVM called the **garbage collector** reclaims its space automatically.

Java has evolved steadily. A new version appears every six months, and every two years one of them is designated a **long-term support** (LTS) release, which vendors keep patched with security and bug fixes for many years. The LTS releases so far are Java 8, 11, 17, 21 and 25; Java 25 was released in September 2025 and is the current LTS release [4]. The newest six-monthly release, Java 27, arrived in September 2026 [5]; it is not an LTS release, so Java 25 remains the version to learn and to deploy. This book targets **Java 25**, and every program in it also compiles and runs on Java 17 and 21, so an older installation at a school laboratory will serve as well.

1.5.2 Why Java: portable, supported, everywhere

Four reasons have kept Java near the top of the programming world for thirty years.

It is portable. A Java program runs on every device that has a Java Virtual Machine — desktop, laptop, server and, through Android's adaptation of the language, smartphone. The slogan is **WORA: Write Once, Run Anywhere**, and Section 1.2.3 showed how bytecode makes it true.

It is supported everywhere. Java has one of the largest communities of any language: free libraries for almost any task, answers to almost any question, and user groups in most large cities, including in Indonesia.

It is widely used, above all for **client–server** systems, in which a browser or phone (the client) talks to a program on a distant computer (the server). Banks, airlines, government services and online shops run enormous Java systems.

It makes object orientation explicit. Every piece of Java code lives inside a class, every variable declares its type, and `public` and `private` say in so many words who may use what. That can feel heavy in a ten-line program, but for a learner it is a gift: the grammar spells out the ideas of Section 1.3.

1.5.3 Python: a short history, and why Python too

Python was created by Guido van Rossum in the Netherlands and first released in 1991 [6]. Its guiding idea is that code is read far more often than it is written, so it should read almost like plain English. Python has no curly braces: the **indentation** of a line — how far it is pushed to the right — shows which block it belongs to. It does not ask you to declare types: a variable can hold any kind of value, and the value knows its own kind. Python checks types while the program runs rather than before, which is called **dynamic typing**; Java checks them when the program is compiled, which is called **static typing**.

Python is fully object-oriented — in Python *every* value, even the number 7, is an object — but it does not force you to write a class before you can print a line. That makes it wonderfully direct for small programs and scripts, and it is why Python became the language of choice for data analysis, scientific computing, automation and, in the last decade, artificial intelligence and machine learning. Python 3.14, released in October 2025, is the current version at the time of writing

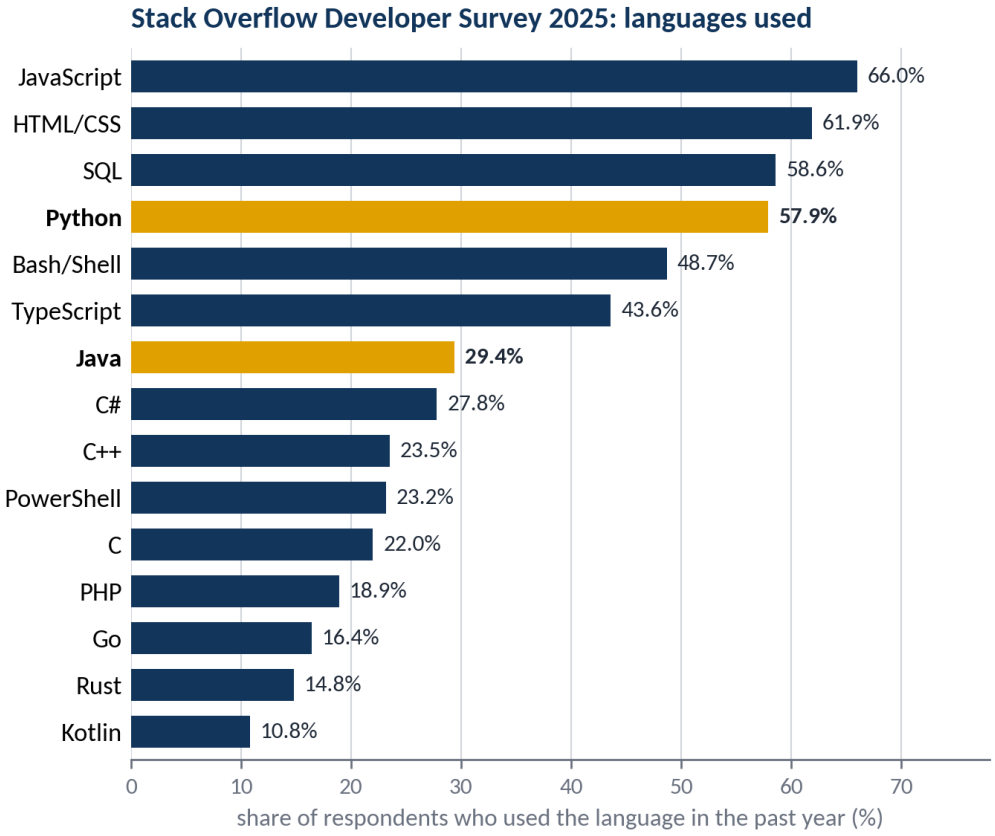
(September 2026), with its seventh maintenance release, 3.14.7, published in August 2026 [7], [8], [9]; this book targets it, and every program in it also runs on Python 3.10 and later.

The reasons for teaching both are practical. Java shows object orientation with every joint visible; Python shows the same ideas with the scaffolding removed. Having seen an idea both ways, you understand the idea and not merely one language's spelling of it. And you will almost certainly use both in your working life.

1.5.4 What the popularity rankings say

"Popular" is a slippery word, and four well-known measurements each tell a different part of the story.

The first is a survey. Every year Stack Overflow, the question-and-answer website for programmers, asks developers around the world which languages they have used in the past year. **Figure 1.5** shows the fifteen most used languages among the 31,771 people who answered that question in the 2025 survey [10]. JavaScript leads at 66.0%, because almost every web page runs it; the markup and query languages HTML/CSS and SQL follow; Python is fourth at 57.9%, and Java seventh at 29.4%, ahead of C#, C++ and C.

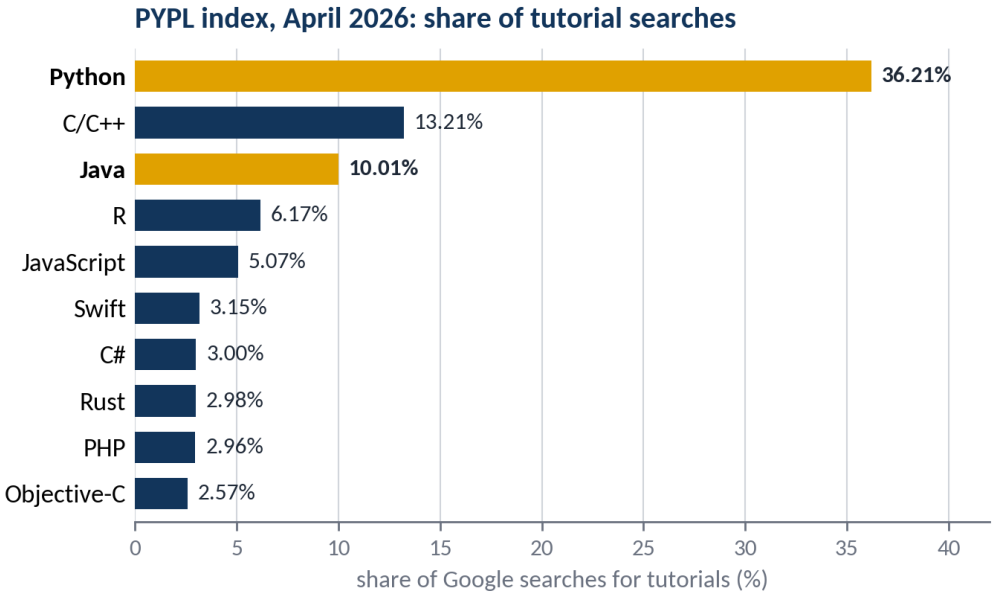


Source: Stack Overflow Developer Survey 2025, all respondents (n = 31,771). Top 15 of the list.

Figure 1.5 The most used languages in the Stack Overflow Developer Survey 2025, all respondents: Python is fourth and Java seventh of the fifteen shown.

The second measurement is the **PYPL index** (PopularitY of Programming Language). It counts how often people search Google for tutorials in each language, using raw data from Google Trends, on the reasoning that the more a language's tutorials are searched for, the more people are learning it [11]. **Figure 1.6** shows its worldwide ranking for April 2026, the latest month we

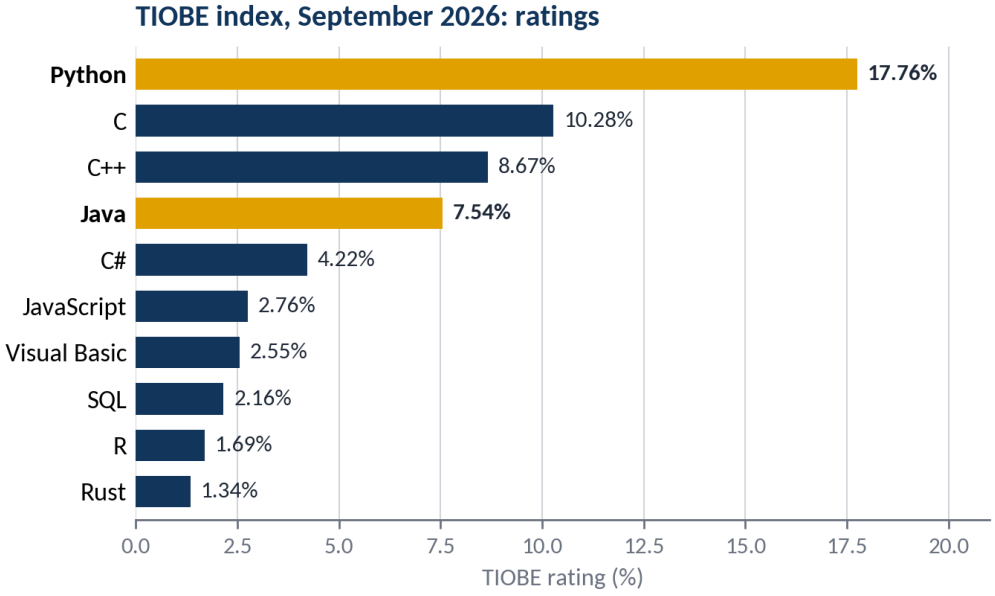
could verify [12]: Python far ahead with 36.21% of tutorial searches, C/C++ second with 13.21%, and Java third with 10.01%.



Source: PYPL PopularitY of Programming Language index, worldwide, April 2026.

Figure 1.6 The PYPL ranking for April 2026, based on how often each language's tutorials are searched for: Python first, Java third.

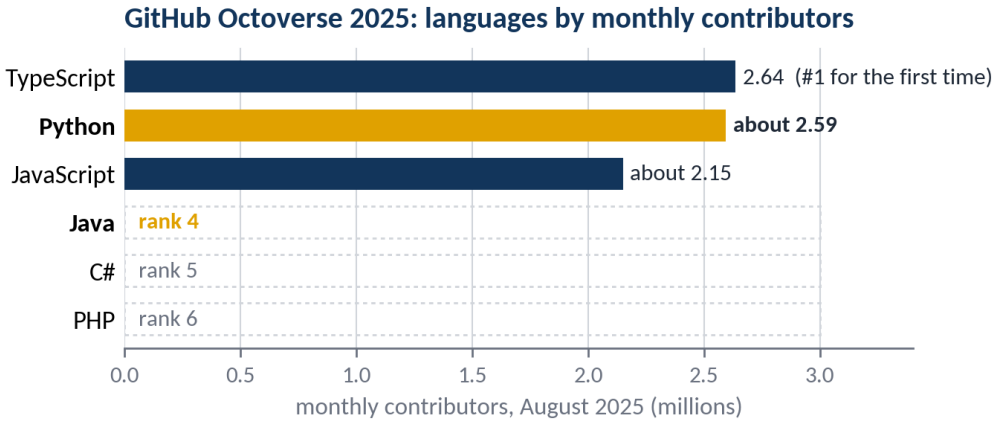
The third is the **TIOBE index**, updated every month. Its ratings are based on the number of skilled engineers world-wide, courses and third-party vendors for each language, gathered through popular search engines and websites such as Google, Bing, Wikipedia, Amazon, YouTube and Baidu [13]. TIOBE itself stresses that its index is not about the *best* programming language, nor about the language in which the most lines of code have been written. **Figure 1.7** shows its ranking for September 2026 [14]: Python first at 17.76%, then C at 10.28%, C++ at 8.67% and Java at 7.54%.



Source: TIOBE Index for September 2026.

Figure 1.7 The TIOBE ranking for September 2026, based on engineers, courses and vendors per language: Python first, Java fourth.

The fourth measures what people actually build in the open. **GitHub** is one of the largest homes of open-source code in the world, and every year its *Octoverse* report counts, among much else, how many developers contribute each month in each language [15]. **Figure 1.8** shows the 2025 ranking. In August 2025 TypeScript, JavaScript with static types added, became the most used language on GitHub for the first time, with 2,636,006 monthly contributors, about 42,000 more than Python; JavaScript was third with about 2.15 million, and Java fourth, followed by C# and PHP.



Source: GitHub Octoverse 2025. Ranks 4-6 are shown by position only.

Figure 1.8 The GitHub Octoverse 2025 ranking by monthly contributors: TypeScript, Python and JavaScript lead, and Java is fourth.

Put the four pictures together and a pattern appears. The measurements disagree about which language is first, because they measure different things — use at work, learning, industry presence, open-source work. Python is at or near the top of every one of them, and Java is in the top seven of every one, in the company of the systems languages C and C++ and the web languages JavaScript and TypeScript. The details shift from month to month, as rankings always do, but the pattern has held for years. A programmer who knows both Java and Python is prepared for most of the software world.

1.5.5 The two languages compared

Table 1.2 sets the two languages next to each other. Do not memorize it now; come back to it whenever you meet one of these differences in a program.

Table 1.2 Java and Python side by side: the differences a beginner meets first.

Question	Java	Python
First released	1995, Sun Microsystems (James Gosling)	1991, Guido van Rossum
Version used in this book	Java 25 LTS (also runs on 17+)	Python 3.14 (also runs on 3.10+)
How it runs	compiled by `javac` to bytecode, run by the JVM (`java`)	run by the interpreter (`python`), which compiles to bytecode internally
Types	static: declared, checked before running	dynamic: checked while running
Blocks of code	curly braces `{ }`	indentation
End of a statement	semicolon `;`	end of the line
Where code lives	every method inside a class	classes optional; plain functions and scripts are fine
Private state	enforced by the keyword `private`	a convention: a leading underscore, `_status`
Printing a line	`System.out.println("Hi");`	`print("Hi")`
Typical uses	large client–server and enterprise systems, Android	data analysis, AI, scripting, web back ends

1.5.6 One way of thinking, two spellings

The most important row of **Table 1.2** is the one that is not there: the *ideas* are the same. A class in Java and a class in Python are both blank forms; an object in either language has state, behavior and identity; sending a message is written with a dot in both. When you learn a new idea in this book, learn it as an idea first, then learn how Java spells it, then how Python spells it. That way, when you meet a third object-oriented language — C#, Kotlin, Swift, TypeScript — you will find that you already know most of it.

1.6 The Tools of the Trade

Before installing anything, let us understand what we are installing and why. A carpenter who knows what each tool in the box does will not panic when one of them is missing.

1.6.1 The JDK and the JRE

To *write* Java programs you need the **Java Development Kit** (JDK). Its official name for the everyday edition is *Java SE Development Kit*, where SE means *Standard Edition*. The JDK contains the compiler `javac`, the launcher `java`, the class library (thousands of ready-made classes for text, files, networking, windows and much more), and a handful of other tools. **Table 1.3** lists the ones this book uses.

Table 1.3 The tools of the Java Development Kit that this book uses.

Tool	What it does	Where this book uses it
`javac`	compiles `.java` source files into `.class` bytecode	from Section 1.7 onward
`java`	starts a JVM and runs a program; also runs a single source file directly	from Section 1.7 onward
`jshell`	an interactive window where you type Java one line at a time	Chapter 3
`javadoc`	builds web-page documentation from comments in the source	Chapter 6
`jar`	packs many `.class` files into one archive that can be shared	Chapter 2

To only *run* Java programs, a smaller package used to be enough: the **Java Runtime Environment** (JRE), with the JVM and the class library but no compiler. For many years the two were installed separately; since Java 11 the JDK contains a complete runtime of its own, so for this book you install the JDK and nothing else.

Besides Oracle's own JDK, several organizations build free, well-tested JDKs from the same OpenJDK source code. This book recommends **Eclipse Temurin**, from the Adoptium project of the Eclipse Foundation [16]: free for any use, available for Windows, macOS and Linux, and regularly updated.

1.6.2 Inside the JDK: the `bin` folder and the PATH

A JDK is a folder, and knowing its layout will save you hours of confusion later. Everything you run lives in its subfolder `bin` (short for *binaries*, meaning runnable programs). **Figure 1.9** looks inside the `bin` folder of a JDK on a Linux computer: `java` runs programs, `javac` compiles them, `javadoc` makes documentation, `jar` packs archives and `jshell` is the interactive window of **Table 1.3**, highlighted in the listing; the other programs are specialist tools you can ignore for now. The JDK in the picture is version 21, but the `bin` folder of JDK 25 holds essentially the same programs, and on Windows each of them carries the extension `.exe`: `java.exe`, `javac.exe`, `javadoc.exe`, and so on. For the examples that follow, suppose that on a Windows computer the JDK was installed in the folder `D:\Program\Java\JDK`, so that its programs live in `D:\Program\Java\JDK\bin`.



Figure 1.9 The `bin` folder of a JDK, listed in a Linux terminal; the five tools of **Table 1.3** are highlighted, and on Windows each file ends in `.exe`.

A **terminal** — on Windows the *Command Prompt* or *PowerShell*, on macOS and Linux the *Terminal* — is a window where you type commands instead of clicking. On Windows you can open the Command Prompt by clicking the Search button on the taskbar and typing `cmd`; Windows suggests *Command Prompt* as the best match, as in **Figure 1.10**, and you click it.

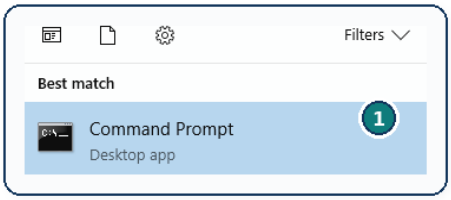


Figure 1.10 Opening the Command Prompt on Windows: after typing `cmd` in the taskbar search box, choose the best match (1).

From a terminal you can do everything: edit a Java program with any plain-text editor (Notepad on Windows, `gedit` or `vim` on Linux), compile it with `javac`, and run it with `java`. But when you type `javac`, how does the terminal know where `javac.exe` is? It looks through a list of folders stored in a setting called the **PATH**. If the JDK's `bin` folder is not on the PATH, the terminal answers that it has never heard of `javac`. For convenience, so that you can call `javac.exe` and `java.exe` from any folder, you add the `bin` folder to the PATH. The quickest way, which lasts only as long as that one terminal window stays open, is the command in **Figure 1.11**.



Figure 1.11 Adding the JDK's `bin` folder to the front of the PATH for the current Command Prompt window with the `set path=` command.

Two remarks about that command. First, the ending `;%path%` matters: `%path%` stands for the old list, so the command puts the JDK's `bin` folder in front of it and keeps everything else. Without it, `set path=D:\Program\Java\JDK\bin` would *replace* the whole PATH of that window with a single folder, and many other commands would stop working in that window. Second, the change disappears when the window closes. A modern installer, such as the Temurin installer of Section 1.7, sets the PATH permanently for you. But whenever a terminal says it does not recognize `java` or `javac`, you now know what has gone wrong: the `bin` folder is not on the PATH of that window.

A related setting, **JAVA_HOME**, names the JDK folder itself (not its `bin`); some tools, such as build systems, read it to find Java. On macOS and Linux the PATH is separated by colons instead of semicolons.

1.6.3 The Python interpreter and its companions

Python is simpler to install because it has a single central program: the **interpreter**, called `python` (on Windows also `py`, and on macOS and Linux usually `python3`). The official distribution from python.org contains the interpreter, the large **standard library** of ready-made modules, a small editor called **IDLE**, and **pip**, the tool that downloads additional libraries from the Python Package Index.

One more Python idea will make your life easier as your projects grow: the **virtual environment**. It is a private folder, usually called `.venv`, holding a copy of the interpreter's settings and its own set of extra libraries for one project, so that two projects needing different versions of a library do not fight. This book's programs use only the standard library and do not need one, but Section 1.7.6 shows how to make one, because VS Code will offer to.

1.6.4 Integrated development environments

You *can* write programs in Notepad and run them from a terminal, and every programmer should do it at least once to understand what happens underneath. But professionals work in an **integrated development environment** (IDE): a single application that combines an editor that understands the language, the compiler or interpreter, a place to see output, and a **debugger** that lets you run a program one line at a time and watch its data change. **Table 1.4** compares the four you are most likely to meet.

Table 1.4 Four development environments for Java and Python.

IDE	Made by	Languages	Notes
Eclipse	Eclipse Foundation, www.eclipse.org	Java first; many others through plug-ins	free; written mostly in Java; a base workspace plus an extensible plug-in system
NetBeans	Apache Software Foundation, netbeans.apache.org	Java; also PHP, C, C++, HTML5, JavaScript	free; built from modules; runs on Windows, macOS, Linux and Solaris
IntelliJ IDEA	JetBrains	Java and Kotlin (PyCharm for Python)	free Community edition; strong refactoring tools
Visual Studio Code	Microsoft	almost any, through extensions	free; light; the same on Windows, macOS and Linux

Eclipse is written mostly in Java and is used above all for Java [17]. It is a base workspace plus a system of **plug-ins** — add-on packages that teach it new tricks — through which it supports dozens of other languages, from C, C++ and PHP (with its CDT and PDT tools) to Python, Rust and Scala, and even LaTeX documents. **NetBeans** [18] builds applications from modular components called **modules**, runs on Windows, macOS, Linux and Solaris, and can itself be extended by other developers.

This book uses **Visual Studio Code** (VS Code) as its main workbench: it is free and the same on all three operating systems, it handles Java and Python equally well in one window, and it runs on a modest laptop. Chapter 2 sets up Eclipse beside it and explores the debugger properly. If your laboratory uses Eclipse, NetBeans or IntelliJ IDEA, every program in this book works there unchanged, because they are all plain `.java` and `.py` files.

1.7 Setting Up VS Code for Java and Python on Windows, macOS, and Linux

Every program in this book is meant to be typed, run and changed by you. Before the next section asks you to do that, let us build the workbench once, carefully, on whichever computer you own. You need four things:

1. **A Java Development Kit**, version 25 LTS. We use **Eclipse Temurin 25** [16]. Any JDK of version 17 or later will run this book's programs, but install 25 if you can.
2. **A Python interpreter**, version 3.14, from python.org or your system's package manager [8]. Any version from 3.10 onward will do.
3. **Visual Studio Code**, the editor, from code.visualstudio.com.
4. **Two VS Code extensions**: the **Extension Pack for Java** and the **Python** extension, both published by Microsoft [19], [20].

Install the language tools first, check them in a terminal, then install the editor, then connect the two. Checking in a terminal *before* opening VS Code is important: if something is wrong with the JDK or Python, a terminal gives a short, clear message, whereas an editor may give a confusing one. **Table 1.5** gathers the whole procedure for the three operating systems on one page, so that you can come back to it later; the subsections after it explain each step.

Table 1.5 The workbench on three operating systems, from an empty computer to running programs in both languages. On Fedora use `sudo dnf install java-25-openjdk-devel python3.14`; on Arch, `sudo pacman -S jdk-openjdk python`.

Step	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
1. Install the JDK	Temurin 25 `.msi` installer from adoptium.net, or `winget install EclipseAdoptium.Temurin.25.JDK`	Temurin 25 `.pkg` installer from adoptium.net, or `brew install --cask temurin@25`	Add the Adoptium repository, then `sudo apt install temurin-25-jdk`
2. Install Python	Installer from python.org (tick **Add python.exe to PATH**), or `winget install Python.Python.3.14`	Installer from python.org, or `brew install python@3.14`	`sudo apt install python3.14 python3.14-venv` (from the *deadsnakes* archive on Ubuntu)
3. Check Java	`java -version` and `javac -version`	the same	the same
4. Check Python	`python --version` (or `py --version`)	`python3 --version`	`python3.14 --version`
5. Install VS Code	Installer from code.visualstudio.com; tick **Add to PATH**	Download, drag *Visual Studio Code* into *Applications*	`.deb` from code.visualstudio.com, or `sudo snap install code --classic`
6. Add the extensions	Extensions view (`Ctrl+Shift+X`): **Extension Pack for Java** and **Python**	the same, with `Cmd+Shift+X`	the same, with `Ctrl+Shift+X`
7. Run a Java file	`javac Hello.java` then `java Hello`	the same	the same
8. Run a Python file	`python hello.py`	`python3 hello.py`	`python3 hello.py`

A note on version numbers before you begin. Java 25 and Python 3.14 receive regular *update releases* that fix bugs and security problems: Java 25.0.1, 25.0.2 and so on, Python 3.14.1, 3.14.2 and so on; in September 2026 the newest were Temurin 25.0.4 and Python 3.14.7 [21], [9]. Always take the newest update of the version the book names; the numbers after the second dot will be larger on your screen than in the examples here, and that is exactly as it should be.

1.7.1 Windows, step by step

Install the JDK.

1. Open a web browser and go to **adoptium.net**. The front page offers the latest LTS release of Temurin for your system; make sure it says **25** and **Windows x64** (or **aarch64** on an ARM laptop), and download the **.msi** installer for the **JDK**.
2. Run the installer. On the page called *Custom Setup*, make sure that **Add to PATH** and **Set JAVA_HOME variable** are both set to be installed — click each one and choose *Will be installed on local hard drive*. These two settings do permanently what **Figure 1.11** did for one window. Accept the rest and click **Install**.
3. If you prefer typing to clicking, Windows 10 and 11 include a package manager called **winget**. Open PowerShell and type `winget install EclipseAdoptium.Temurin.25.JDK`; it downloads and runs the same installer.

Install Python.

4. Go to **python.org**, open **Downloads → Windows**, and download Python **3.14**. The download page offers the *Python install manager*, which installs Python and a `py` command that can manage several versions side by side, as well as a classic installer. Either is fine. With the classic installer, **tick the box "Add python.exe to PATH"** on its first page before clicking *Install Now* — this is the single most common thing beginners miss. With the install manager, follow its prompts, then type `py install 3.14` in a new terminal if it asks you to choose a version.
5. Alternatively, in PowerShell: `winget install Python.Python.3.14`.

Check both.

6. Close every terminal window you have open, and open a *new* PowerShell or Command Prompt window. Old windows keep the old PATH and will not find the new tools. Type these three commands, pressing Enter after each:

```
java -version
javac -version
python --version
```

You should see something close to the following. The first command prints three lines about the runtime, the second prints the compiler's version, and the third prints Python's.

```
openjdk version "25.0.1" 2025-10-21 LTS
OpenJDK Runtime Environment Temurin-25.0.1+8 (build 25.0.1+8-LTS)
OpenJDK 64-Bit Server VM Temurin-25.0.1+8 (build 25.0.1+8-LTS, mixed mode, sharing)
javac 25.0.1
Python 3.14.0
```

If `python --version` opens the Microsoft Store instead of printing a version, or prints nothing at all, Windows is running a small placeholder program called an *app execution alias* instead of the real Python. Try `py --version`; if that works, use `py` wherever this book says `python`. To make `python` itself work, open **Settings → Apps → Advanced app settings → App execution aliases** and switch off the two entries named *python.exe* and *python3.exe*, then open a new terminal.

1.7.2 macOS, step by step

Install the JDK.

1. Go to **adoptium.net**, choose **Temurin 25**, **macOS**, and the right processor: **aarch64** for a Mac with Apple silicon (M1, M2, M3, M4 and later), **x64** for an older Intel Mac. If you are not sure, open the Apple menu → **About This Mac**: it says *Chip Apple M…* or *Processor Intel…*. Download the **.pkg** installer and run it. It installs the JDK into `/Library/Java/JavaVirtualMachines/`, where macOS finds it automatically; there is no PATH to set.
2. If you use the **Homebrew** package manager, `brew install --cask temurin@25` does the same.

Install Python.

3. macOS may already have a `python3` command, but it is often an old version that belongs to the system's own developer tools; do not rely on it. Go to **python.org → Downloads → macOS**, download the **Python 3.14** installer (a *universal2* package that works on both kinds of Mac), and run it. At the end, the installer opens a folder; double-click **Install Certificates.command** in it, so that Python can make secure web connections.
4. With Homebrew: `brew install python@3.14`.

Check both. Open **Terminal** (press `Cmd+Space`, type *Terminal*, press Enter) and type:

```
java -version
javac -version
python3 --version
```

On macOS, type `python3`, not `python`: the name `python` is either missing or points somewhere unhelpful. Everywhere this book writes `python`, a Mac user types `python3`.

1.7.3 Linux, step by step

Linux distributions install software from **repositories** through a package manager. The commands below are for Ubuntu and Debian; **Table 1.5** gives the equivalents for Fedora and Arch.

Install the JDK. Your distribution may already offer a package called `openjdk-25-jdk`; try `sudo apt install openjdk-25-jdk` first. If it is not found, add the Adoptium repository, which always carries the current Temurin releases. Type these lines one at a time; the long ones are single lines:

```
sudo apt install -y wget apt-transport-https gpg
wget -qO - https://packages.adoptium.net/artifactory/api/gpg/key/public | gpg --dearmor |
sudo tee /etc/apt/trusted.gpg.d/adoptium.gpg > /dev/null
echo "deb https://packages.adoptium.net/artifactory/deb $(awk -F=
'/^VERSION_CODENAME/{print$2}' /etc/os-release) main" | sudo tee
/etc/apt/sources.list.d/adoptium.list
sudo apt update
sudo apt install temurin-25-jdk
```

The first line installs three small helper programs; the second downloads the key with which Adoptium signs its packages, so that your system can check they are genuine; the third tells the package manager where the repository is; the last two refresh the package list and install the JDK. Type your password when `sudo` asks for it — nothing appears on the screen as you type, which is normal.

Install Python. Ubuntu ships Python 3 as part of the system, but usually not the newest version, and you must never remove or replace the system's own Python, because the operating system itself uses it. Install 3.14 *beside* it. On Ubuntu the usual source is the *deadsnakes* archive:

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt update
sudo apt install python3.14 python3.14-venv
```

Then `python3.14` runs the new version and `python3` still runs the system's version. If your distribution already has Python 3.10 or later as `python3` — check with `python3 --version` — you may simply use that for this book.

Check both.

```
java -version
javac -version
python3.14 --version
```

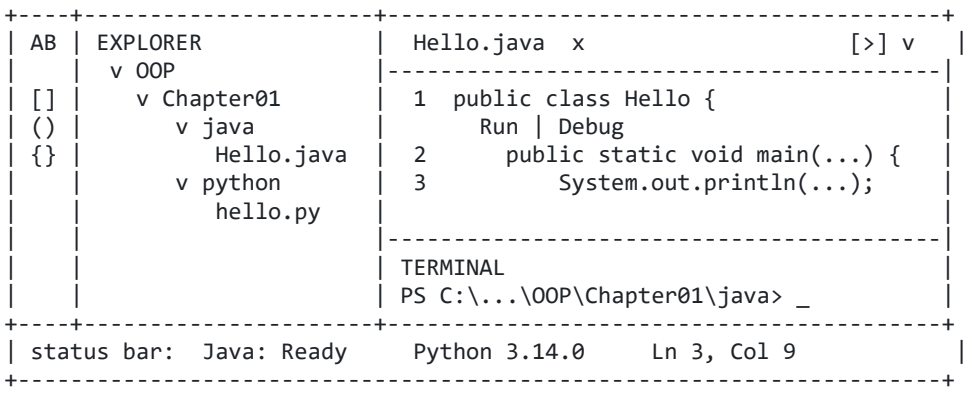
If `java -version` reports an older Java that was already installed, choose the new one with `sudo update-alternatives --config java` and again for `javac`.

1.7.4 VS Code and its two extensions

1. **Install VS Code.** Download it from **code.visualstudio.com**. On Windows, run the installer and, on the page *Select Additional Tasks*, tick **Add to PATH** and the two **Open with Code** options; they let you type `code .` in a terminal, or right-click a folder in Explorer, to open that folder in VS Code. On macOS, open the downloaded archive and drag *Visual Studio Code* into *Applications*; then start VS Code, press `Cmd+Shift+P` to open the **Command Palette**, type *shell command* and choose **Shell Command: Install 'code' command in PATH**. On Linux, install the `.deb` package with `sudo apt install ./code_*.deb` from the download folder, or use `sudo snap install code --classic`.

2. **Install the Java extensions.** Start VS Code and open the **Extensions** view: click the icon of four small squares on the left edge, or press `Ctrl+Shift+X` (`Cmd+Shift+X` on a Mac). Search for **Extension Pack for Java** and install the one published by Microsoft. It is a bundle that installs several extensions at once, among them *Language Support for Java by Red Hat* (which understands Java code, colors it, completes it and underlines mistakes as you type), *Debugger for Java*, *Test Runner for Java* (used in Chapter 6), *Maven for Java* and *Project Manager for Java* [19].
3. **Install the Python extension.** In the same view, search for **Python** and install the one published by Microsoft. It also brings **Pylance**, which understands Python code, and the **Python Debugger** [20].
4. **Tell VS Code which Java and which Python to use.** Usually it finds them by itself. To check, open the Command Palette (`Ctrl+Shift+P` or `Cmd+Shift+P`). For Java, run **Java: Configure Java Runtime**: the page that opens lists the JDKs VS Code has found; make sure the one for your project is 25. For Python, run **Python: Select Interpreter** and choose Python 3.14 from the list. The chosen interpreter is shown in the status bar at the bottom right whenever a Python file is open.
5. **Make a home for this book.** Create a folder, for example `OOP` inside your Documents folder, with one subfolder per chapter and inside each a `java` and a `python` folder: `OOP/Chapter01/java`, `OOP/Chapter01/python`. This is exactly how the accompanying code folder `Code_new` is organized. In VS Code choose **File → Open Folder** and open `OOP`. If VS Code asks whether you trust the files in this folder, answer yes: it is your own folder.

The VS Code window has five areas you will use constantly. Here is a sketch, so that the names in the following steps mean something:



The **activity bar** (AB) on the far left switches between views: Explorer (your files), Search, Source Control, Run and Debug, and Extensions. The **side bar** next to it shows the current view — here the Explorer with the folder tree. The **editor** in the middle holds your open files, with the **Run** button (a triangle) at its top right. The **panel** below the editor holds the **Terminal**, a real terminal already opened in your folder; open it with **Terminal → New Terminal**. The **status bar** at the bottom tells you, among other things, whether the Java support has finished starting ("Java: Ready") and which Python interpreter is selected.

1.7.5 Your first Java program

In the Explorer, right-click the folder `Chapter01/java`, choose **New File**, and name it `Hello.java` — with a capital H, because in Java the file name must match the class name exactly, capitals included. Type in **Listing 1.3** and save it with `Ctrl+S` (`Cmd+S`).

Listing 1.3 `Hello.java`: the first Java program, which greets the reader and reports which Java version is running it.

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello from Profitina Laundry!");
        System.out.println("Java is running on version " +
System.getProperty("java.version") + ".");
    }
}
```

There are three ways to run it, and you should try all three once.

With the Run button. Above the line `public static void main`, VS Code shows two small links, **Run | Debug**. Click **Run**. (Or click the triangle at the top right of the editor.) VS Code compiles the file and runs it, and the output appears in the Terminal panel.

With two commands in the terminal, exactly as **Figure 1.1** describes. Open the terminal (**Terminal → New Terminal**), move into the folder with `cd Chapter01/java` (on Windows `cd Chapter01\java`), and type:

```
javac Hello.java
java Hello
```

The first command compiles `Hello.java` into `Hello.class` and prints nothing if all is well — in programming, silence usually means success. The second runs the class. Notice that you write `java Hello`, without `.class`: you are naming the class to run, not the file.

With one command. Since Java 11 the launcher can compile a single source file in memory and run it at once [22]: `java Hello.java`. This is handy for small programs; for programs made of several files, such as the `Order` program of Section 1.8, use `javac` first.

Whichever way you choose, the program prints:

```
Hello from Profitina Laundry!
Java is running on version 21.0.10.
```

The second line tells you which Java actually ran the program. The listings in this book were checked on the Java 21 of the machine that produced the book, which is why it says `21.0.10`; on your computer, if you followed Section 1.7.1, 1.7.2 or 1.7.3, it will say `25.0.` followed by the update number. If it shows some older number, VS Code or the terminal is using a different JDK than the one you installed; Section 1.7.8 tells you how to fix it.

Now let us read **Listing 1.3** line by line.

- `public class Hello {` declares a class named `Hello`. In Java every piece of code lives inside a class, even a program that has nothing to do with objects yet. The word `public` means that any other code may use the class. The opening brace `{` begins the class's body, which ends at the matching `}` on the last line.
- `public static void main(String[] args) {` declares the **main method**, the place where every Java program starts. Each word has a job. `public`: the Java launcher, which is outside the class, must be allowed to call it. `static`: it belongs to the class itself, so it can run before any object of the class exists. `void`: it returns no answer. `main`: the name the launcher looks for. `String[] args`: an array of strings holding any words typed after the class name on the command line; we do not use it yet, but it must be there.
- `System.out.println("Hello from Profitina Laundry!");` prints a line. `System` is a class from the Java library; `out` is an object inside it that represents the terminal's output; `println` ("print

line") is a message sent to that object, asking it to print the text in quotes and then move to a new line. The semicolon ends the statement.

- The next statement builds its line with `+`, which between strings means *join together*: the text `"Java is running on version "`, then the value returned by `System.getProperty("java.version")` — a message asking the running system for its Java version — then a full stop.

> **KEY INSIGHT.** Even the first program is full of objects and messages: `System.out` is an object, and `println(...)` is a message sent to it with a dot. You have been doing object-oriented programming since your first line of Java.

Java 25 adds a lighter way to write a first program, called a *compact source file*: a file may contain just `void main() { IO.println("Hello"); }` with no class around it [23]. It is a pleasant novelty for beginners, but it does not exist in Java 17 or 21, so this book always writes the full form of **Listing 1.3**, which works on every version.

1.7.6 Your first Python program

In the Explorer, right-click `Chapter01/python`, choose **New File**, name it `hello.py` (Python file names are conventionally all lowercase), and type in **Listing 1.4**.

Listing 1.4 `hello.py`: the first Python program, which does the same as **Listing 1.3**.

```
import platform

print("Hello from Profitina Laundry!")
print("Python is running on version " + platform.python_version() + ".")
```

Run it one of two ways.

With the Run button. Click the triangle at the top right of the editor (its tooltip says **Run Python File**). VS Code runs the file with the interpreter shown in the status bar, and the output appears in the Terminal panel.

In the terminal. Move into the folder (`cd Chapter01/python`) and type `python hello.py` on Windows, `python3 hello.py` on macOS and Linux (or `python3.14 hello.py` if you installed 3.14 beside an older system Python). There is no separate compile step.

The program prints:

```
Hello from Profitina Laundry!
Python is running on version 3.11.15.
```

Again the second line is honest about which interpreter ran the program: the book's machine has Python 3.11, and yours will say `3.14.` and an update number.

Line by line:

- `import platform` makes the standard library's `platform` module available. A **module** is a file of ready-made Python code; `import` brings it into your program. Java needed no such line because `System` is always available.
- The empty line is only for the reader's eyes.
- `print("Hello from Profitina Laundry!")` prints a line. `print` is a built-in function; there is no class to write, no `main`, no semicolon. The whole file is the program, run from top to bottom.
- `print("Python is running on version " + platform.python_version() + ".")` joins three strings with `+`, just as Java did. `platform.python_version()` asks the module for the running version.

Set the two listings side by side and you have the whole of **Table 1.2** in miniature: Java wraps its program in a class and a `main` method and ends statements with semicolons; Python runs the file from the top; both join text with `+`; both use a dot to reach inside something (`System.out`, `platform.python_version`).

A virtual environment, if VS Code offers one. When you open a Python folder, VS Code may suggest creating a virtual environment. For this book it is optional. If you want one, open the Command Palette and run **Python: Create Environment**, choose **Venv** and your Python 3.14; VS Code creates a `.venv` folder and selects it. The terminal equivalent is `python -m venv .venv` (or `python3 -m venv .venv`).

1.7.7 Debugging: watching a program think

When a program gives a wrong answer, the fastest way to find out why is to *watch* it run, one line at a time, and look at its data as it changes. The tool that lets you do that is the **debugger**. Let us try it on a program that has a loop in it, so that there is something to watch. Profitina Laundry's machines each hold $c = 8$ kilograms; an order of w kilograms therefore needs

$$L(w) = \left\lceil \frac{w}{c} \right\rceil \tag{1.3}$$

loads, where $\lceil x \rceil$, the "ceiling of x", is the smallest whole number not less than x: a 9-kilogram order does not fit in one machine, so it needs $\lceil 9/8 \rceil = \lceil 1.125 \rceil = 2$ loads. Computers divide whole numbers by throwing away the remainder, which is a floor, not a ceiling; the standard trick for getting a ceiling out of whole-number division is

$$\left\lceil \frac{w}{c} \right\rceil = \left\lfloor \frac{w + c - 1}{c} \right\rfloor \quad \text{for whole numbers } w \geq 0,\ c \geq 1. \tag{1.4}$$

Why is Equation (1.4) true? Write $w = qc + r$ with remainder $0 \leq r \leq c - 1$. If $r = 0$, then $w/c = q$ exactly, and $(w + c - 1)/c = q + (c - 1)/c$, whose floor is q because $(c - 1)/c < 1$. If $r \geq 1$, the ceiling is $q + 1$, and $(w + c - 1)/c = q + (r + c - 1)/c$, where $c \leq r + c - 1 \leq 2c - 2 < 2c$, so the floor is $q + 1$. Both cases agree. Check with the numbers: $w = 9$, $c = 8$ gives $\lfloor 16/8 \rfloor = 2$; $w = 8$ gives $\lfloor 15/8 \rfloor = 1$; $w = 3$ gives $\lfloor 10/8 \rfloor = 1$.

Listing 1.5 and **Listing 1.6** apply Equation (1.4) to each of five orders and add up the loads.

Listing 1.5 `Loads.java`: counting the machine loads for today's orders, a small program with a loop to watch in the debugger.

```
public class Loads {
    public static void main(String[] args) {
        int capacity = 8;                  // kilograms one machine washes at once
        int[] orders = {3, 9, 5, 12, 7};   // kilograms in each order today
        int totalLoads = 0;

        for (int i = 0; i < orders.length; i++) {
            // integer ceiling: how many full or partial loads this order needs
            int loads = (orders[i] + capacity - 1) / capacity;
            totalLoads = totalLoads + loads;
            System.out.println("order " + (i + 1) + ": " + orders[i]
                    + " kg -> " + loads + " load(s)");
        }
        System.out.println("machine loads today: " + totalLoads);
    }
}
```

Listing 1.6 `loads.py`: the same count in Python; `//` is Python's whole-number division.

```
capacity = 8                  # kilograms one machine washes at once
orders = [3, 9, 5, 12, 7]     # kilograms in each order today
total_loads = 0

for i in range(len(orders)):
    # integer ceiling: how many full or partial loads this order needs
    loads = (orders[i] + capacity - 1) // capacity
    total_loads = total_loads + loads
    print("order " + str(i + 1) + ": " + str(orders[i])
          + " kg -> " + str(loads) + " load(s)")
print("machine loads today: " + str(total_loads))
```

Both print:

```
order 1: 3 kg -> 1 load(s)
order 2: 9 kg -> 2 load(s)
order 3: 5 kg -> 1 load(s)
order 4: 12 kg -> 2 load(s)
order 5: 7 kg -> 1 load(s)
machine loads today: 7
```

Check by hand with Equation (1.3): $\lceil 3/8 \rceil = 1$, $\lceil 9/8 \rceil = 2$, $\lceil 5/8 \rceil = 1$, $\lceil 12/8 \rceil = 2$, $\lceil 7/8 \rceil = 1$, and $1 + 2 + 1 + 2 + 1 = 7$.

Now debug it. The steps are the same in both languages:

1. **Set a breakpoint.** In `Loads.java`, move the mouse to the left of the line number of the line `totalLoads = totalLoads + loads;` and click. A red dot appears: a **breakpoint**, a place where the debugger will pause the program.
2. **Start the debugger.** Click **Debug** above `main` (Java), or press `F5`. For Python, open `loads.py`, set a breakpoint on `total_loads = total_loads + loads`, press `F5`, and choose **Python Debugger: Current File** if VS Code asks. The program starts and stops at the red dot, with that line highlighted in yellow. The highlighted line has *not yet* run.
3. **Look at the Variables panel** on the left. It lists every variable and its current value: `capacity = 8`, `orders` (click the arrow to see its five numbers), `totalLoads = 0`, `i = 0`, `loads = 1`.
4. **Step.** Press `F10` (**Step Over**) to run the highlighted line and stop at the next one. Watch `totalLoads` change from `0` to `1`. Press `F5` (**Continue**) to run until the breakpoint is reached again, in the next pass of the loop: now `i = 1`, `loads = 2`, `totalLoads = 1`. Continue again, and again, and watch the total grow: 1,3,4,6,7.
5. **Watch an expression.** In the **Watch** panel, click `+` and type `orders[i] / capacity`; the same expression works in both languages. In Java, dividing two `int` values gives a whole number, so for `i = 1` the watch shows `1`; in Python, `/` always gives a fraction and shows `1.125`. That single difference is why the program uses `(orders[i] + capacity - 1) / capacity` in Java and `//` in Python.
6. **Stop.** Press `Shift+F5`, or click the red square in the small debugging toolbar at the top of the window.

The other keys on that toolbar are worth knowing from the start: `F11` (**Step Into**) goes *inside* a method call on the current line, which you will use constantly once programs are made of many methods; `Shift+F11` (**Step Out**) runs to the end of the current method and returns to its caller. Chapter 2 is devoted to debugging and uses all of them.

1.7.8 When something goes wrong

Every installation problem a beginner meets has been met by thousands before. **Table 1.6** lists the common ones, what they look like and how to fix them.

Table 1.6 Common problems when setting up Java, Python and VS Code, and how to fix them.

What you see	What it means	What to do
`'java' is not recognized as an internal or external command` (Windows), or `java: command not found`	the JDK's `bin` folder is not on the PATH of this terminal	open a *new* terminal; if it persists, reinstall with **Add to PATH** ticked, or add the `bin` folder by hand (Section 1.6.2)
`java -version` shows 25 but `javac -version` is not found	a runtime without the compiler is first on the PATH	install the **JDK**, not only a JRE; put the JDK's `bin` first on the PATH
`java -version` shows an old version such as 1.8	an older Java is earlier on the PATH	remove the old one, or move the new `bin` folder to the top of the PATH; on Linux use `update-alternatives`
`error: class Hello is public, should be declared in a file named Hello.java`	the file name and the class name differ, perhaps only in capitals	rename the file to match the class exactly: `Hello.java` for `class Hello`
`Error: Could not find or load main class Hello.class`	you typed `java Hello.class`	type `java Hello`, naming the class, not the file
`python` opens the Microsoft Store, or prints nothing (Windows)	an app execution alias is standing in for Python	use `py`, or switch off the aliases (Section 1.7.1) and open a new terminal
`python: command not found` (macOS, Linux)	on these systems the command is `python3`	type `python3` (or `python3.14`)
`IndentationError: unexpected indent`	a Python line is pushed right when it should not be, or tabs and spaces are mixed	line up the block with spaces only; VS Code inserts four spaces per `Tab` by default
VS Code runs Python 3.11 (or another version) instead of 3.14	another interpreter is selected	**Python: Select Interpreter** and pick 3.14; check the status bar
Java code in VS Code shows no colors or completions, status bar shows a spinning "Java"	the Java language server is still starting, or no JDK 17+ was found for it	wait a minute on first start; run **Java: Configure Java Runtime** and make sure a JDK is listed
The program window flashes and closes	you double-clicked a file instead of running it from VS Code or a terminal	run it with the Run button or from the terminal

With the workbench standing, every listing in the rest of this book is one copy and one run away from being yours.

1.8 Your First Class: a Laundry Order

The workbench is ready and both languages have said hello. Now we build the right-hand side of **Figure 1.4**: an `Order` class that bundles an order's state with its behavior and protects the one with the other. We approach it in three steps. First a plain program with variables, to see how values are named and computed. Then the class. Then a small program that creates orders and sends them messages.

1.8.1 A receipt: variables and arithmetic

A **variable** is a named box that holds a value. **Listing 1.7** and **Listing 1.8** compute Ani's receipt with a handful of them.

Listing 1.7 `Receipt.java`: a receipt computed with variables of four types.

```java
public class Receipt {
    public static void main(String[] args) {
        String customer = "Ani";
        double kilograms = 3.5;
        long pricePerKg = 7000;
        long price = Math.round(kilograms * pricePerKg);
        long paid = 50000;
        long change = paid - price;

        System.out.println("PROFITINA LAUNDRY - RECEIPT");
        System.out.println("Customer : " + customer);
        System.out.println("Weight   : " + kilograms + " kg");
        System.out.println("Price    : Rp " + price);
        System.out.println("Paid     : Rp " + paid);
        System.out.println("Change   : Rp " + change);
    }
}
```

Listing 1.8 `receipt.py`: the same receipt in Python, where the values decide their own types.

```python
customer = "Ani"
kilograms = 3.5
price_per_kg = 7000
price = round(kilograms * price_per_kg)
paid = 50000
change = paid - price

print("PROFITINA LAUNDRY - RECEIPT")
print("Customer : " + customer)
print("Weight   : " + str(kilograms) + " kg")
print("Price    : Rp " + str(price))
print("Paid     : Rp " + str(paid))
print("Change   : Rp " + str(change))
```

Both print:

```
PROFITINA LAUNDRY - RECEIPT
Customer : Ani
Weight   : 3.5 kg
Price    : Rp 24500
Paid     : Rp 50000
Change   : Rp 25500
```

In Java, each variable is **declared** with its type the first time it appears. `String customer = "Ani";` makes a box called `customer` that may hold only text, and puts `"Ani"` in it. `double kilograms = 3.5;` makes a box for a number with a fractional part. `long pricePerKg = 7000;` makes a box for a whole number; Java has two common whole-number types, `int`, which holds values up to about 2.1 billion, and `long`, which holds values up to about 9.2×10^{18}, and we use `long` for money so that a large total can never overflow. `Math.round(kilograms * pricePerKg)` multiplies and rounds to the nearest whole number, exactly as Equation (1.1) says; `Math` is a library class that holds mathematical functions.

In Python there are no type words: `kilograms = 3.5` creates a variable that happens to hold a fractional number, and `price_per_kg = 7000` one that happens to hold a whole number. Python's whole numbers have no upper limit at all. The function `round` rounds, and `str(...)` converts a number into text so that it can be joined to other text with `+`; in Java, joining a number to a string converts it automatically, but Python insists that you ask. Notice also the naming style: Java writes multi-word names in **camelCase** (`pricePerKg`), Python in **snake_case** (`price_per_kg`). Each community follows its own convention, and this book follows both.

The change is $50000 - 24500 = 25500$, and the program agrees. So far, though, nothing here is an object of our own. The receipt is five loose variables. Let us bundle them.

1.8.2 Designing the `Order` class

Before writing code, we design, using the questions of Section 1.3.

What does an order know? Its **number**, the **customer**'s name, the **kilograms**, whether it is **express**, and its **status**. The first four are fixed once the ticket is written; the status changes as the laundry moves through the shop.

What can an order do? Tell its **price**, computed by Equation (1.1) and Equation (1.2) from its own weight and express flag. Move its status along: **start washing**, **mark ready**, **pick up**. Report its **status**. And describe itself in one line, for printing.

What must never happen? The status must move only forward, one step at a time, in the order RECEIVED, WASHING, READY, PICKED UP. In particular, an order must never be picked up before it is ready. **Figure 1.12** draws these rules as a **state diagram**: each rounded box is a state, and each arrow is a message that moves the order from one state to the next.

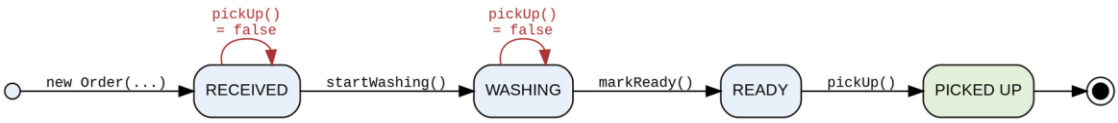


Figure 1.12 The life of an order: each message moves it one step forward, and a message sent at the wrong moment, such as `pickUp()` while the laundry is still washing, is refused and changes nothing.

The red loops in **Figure 1.12** are as important as the black arrows. They say that `pickUp()` sent to a RECEIVED or WASHING order returns `false` and leaves the status exactly as it was. The same is true of every other message sent at the wrong moment; the figure draws only the most tempting one. An order that *refuses* nonsense is the whole point of this section.

1.8.3 The `Order` class in Java

Listing 1.9 is the class. It is the longest program so far, so we read it slowly afterward.

Listing 1.9 `Order.java`: a laundry order that holds its own state, computes its own price, and moves its status forward only when that makes sense.

```
public class Order {
    // State: what every order knows about itself.
    private final int number;
    private final String customer;
    private final double kilograms;
    private final boolean express;
    private String status;

    // Constructor: runs once, when a new order is created with "new".
    public Order(int number, String customer, double kilograms, boolean express) {
        this.number = number;
        this.customer = customer;
        this.kilograms = kilograms;
        this.express = express;
        this.status = "RECEIVED";
    }

    // Behavior: what every order can do when it receives a message.
    public long price() {
        long base = Math.round(kilograms * 7000);   // Rp 7,000 per kilogram
        if (express) {
            return base + base / 2;                 // express costs 50% more
        }
        return base;
    }

    public boolean startWashing() {
        if (!status.equals("RECEIVED")) {
            return false;                           // refuse: wrong moment
        }
        status = "WASHING";
        return true;
    }

    public boolean markReady() {
        if (!status.equals("WASHING")) {
            return false;
        }
        status = "READY";
        return true;
    }
```

Listing 1.9 (Continued) *Order.java: a laundry order that holds its own state, computes its own price, and moves its status forward only when that makes sense.*

```
    public boolean pickUp() {
        if (!status.equals("READY")) {
            return false;                           // cannot collect wet laundry
        }
        status = "PICKED UP";
        return true;
    }

    public String getStatus() {
        return status;
    }

    @Override
    public String toString() {
        String kind = express ? " express" : "";
        return "Order #" + number + " for " + customer + ": " + kilograms + " kg"
                + kind + ", Rp " + price() + ", " + status;
    }
}
```

This file has no `main` method, so it cannot be run by itself: it is a blank form, not a program. Its parts, from top to bottom:

Fields. The five lines beginning `private` declare the state that every `Order` object will carry, matching the class box of **Figure 1.2**. `private` means that no code outside the class `Order` may read or write these fields directly: a line such as `ani.status = "PICKED UP";` written in another class is rejected by the compiler with the message *status has private access in Order*. That single word turns the rule "only the order changes its status" from a hope into a guarantee. The word `final` on the first four fields means that once they have been given a value, in the constructor, they can never change. `status` is not `final`, because it must change.

Constructor. `public Order(int number, String customer, double kilograms, boolean express)` looks like a method but has no return type and has the same name as the class. It runs exactly once for each new object, at the moment the object is created, and its job is to fill in the blank form. The four values in parentheses are its **parameters**: the information the caller supplies. Inside, `this.number = number;` copies the parameter `number` into the field `number` of the object being created. The word `this` means *this very object*, and it is needed here because the parameter and the field have the same name; `this.number` is the field, plain `number` is the parameter. The last line sets the status to `"RECEIVED"`, because every order starts there — the caller cannot choose to create an order that is already picked up.

The `price` method. `public long price()` takes no parameters, because everything it needs is already inside the object, and returns a `long`. It computes `base` with Equation (1.1). The `if (express)` statement adds half of `base` when the express flag is true; `base / 2` divides two whole numbers, which in Java discards the remainder — exactly the floor of Equation (1.2). The keyword `return` ends the method and hands the value back to whoever sent the message.

The three status methods. `startWashing()`, `markReady()` and `pickUp()` all follow the same pattern: *check, then act*. `pickUp()`, for example, first asks `!status.equals("READY")` — "is the status *not* READY?" (the `!` means *not*). If so, the order refuses: `return false;` ends the method at once, and the status is untouched. Otherwise it sets the status to `"PICKED UP"` and returns `true`, so that the caller knows the request succeeded. Note that strings are compared with `equals`, not with `==`; Section 1.11 explains why that matters in Java.

`getStatus()` hands out the status for reading. Outsiders may *look* at the status through this method, but there is no method that lets them *set* it to an arbitrary value. A method like this, which only reports a field, is called a **getter**.

`toString()` returns a one-line description of the order. It is special in Java: whenever an object is printed with `System.out.println` or joined to a string with `+`, Java calls its `toString()` method automatically to find out what to show. The line `@Override` above it tells the compiler that we are deliberately replacing a `toString()` that every Java object already has — a dull default that prints something like `Order@1b6d3586`. Inside, `express ? " express" : ""` is the **conditional operator**: it gives `" express"` if `express` is true and the empty string otherwise.

1.8.4 The `Order` class in Python

Listing 1.10 is the same class in Python.

Listing 1.10 `order.py`: the same `Order` class in Python.

```
class Order:
    # Constructor: runs once, when a new order is created with Order(...).
    def __init__(self, number, customer, kilograms, express):
        # State: what every order knows about itself.
        self._number = number
        self._customer = customer
        self._kilograms = kilograms
        self._express = express
        self._status = "RECEIVED"

    # Behavior: what every order can do when it receives a message.
    def price(self):
        base = round(self._kilograms * 7000)   # Rp 7,000 per kilogram
        if self._express:
            return base + base // 2            # express costs 50% more
        return base

    def start_washing(self):
        if self._status != "RECEIVED":
            return False                       # refuse: wrong moment
        self._status = "WASHING"
        return True

    def mark_ready(self):
        if self._status != "WASHING":
            return False
        self._status = "READY"
        return True

    def pick_up(self):
        if self._status != "READY":
            return False                       # cannot collect wet laundry
        self._status = "PICKED UP"
        return True

    def get_status(self):
        return self._status

    def __str__(self):
        kind = " express" if self._express else ""
        return ("Order #" + str(self._number) + " for " + self._customer + ": "
                + str(self._kilograms) + " kg" + kind + ", Rp "
                + str(self.price()) + ", " + self._status)
```

The ideas are identical; the spelling differs in five places.

1. **The class line** is `class Order:`, with a colon, and everything indented beneath it belongs to the class. There are no braces anywhere.
2. **The constructor** is the method with the odd name `__init__` (two underscores before and after; say "dunder init"). Python calls it automatically when an object is created. Its first parameter, `self`, is the object being created — the same idea as Java's `this`, but in Python you must write it explicitly as the first parameter of every method.
3. **The fields are not declared in advance.** They come into existence when the constructor first assigns to them: `self._status = "RECEIVED"` creates the field `_status` on the new object.
4. **Privacy is a convention.** Python has no `private` keyword. A leading underscore, as in `_status`, is the community's agreed signal meaning "this belongs to the object; do not touch it from outside". Python will not stop a determined programmer from writing `ani._status = "PICKED UP"`, but every Python programmer who reads that line knows it breaks the rule, and tools such as Pylance in VS Code flag it. Chapter 7 shows Python's stronger options.

5. **Printing uses `__str__`.** The method Python calls when an object is printed is `__str__`, the counterpart of Java's `toString()`. And the conditional expression is written in words: `" express" if self._express else ""`.

Two smaller differences are worth noticing. Python's whole-number division is `//`, so `base // 2` is the floor of Equation (1.2) directly, and methods are named in snake_case: `start_washing`, `mark_ready`, `pick_up`, `get_status`. And Python compares strings with `!=` and `==` directly, which in Python compare the *contents* of the strings — safe here, unlike Java's `==`.

1.8.5 Bringing orders to life

A class does nothing until somebody creates objects from it and sends them messages. **Listing 1.11** and **Listing 1.12** are the programs that do.

Listing 1.11 `OrderDemo.java`: creating two orders and sending them messages, including a premature pick-up that is refused.

```
public class OrderDemo {
    public static void main(String[] args) {
        Order ani = new Order(1, "Ani", 3.5, false);
        Order budi = new Order(2, "Budi", 5.0, true);
        System.out.println(ani);
        System.out.println(budi);

        System.out.println("Ani tries to pick up at once: " + ani.pickUp());
        ani.startWashing();
        ani.markReady();
        System.out.println("After washing and drying:     " + ani.getStatus());
        System.out.println("Ani picks up now: " + ani.pickUp());

        System.out.println(ani);
        System.out.println(budi);
    }
}
```

Listing 1.12 `order_demo.py`: the same program in Python, importing the class from `order.py`.

```
from order import Order

ani = Order(1, "Ani", 3.5, False)
budi = Order(2, "Budi", 5.0, True)
print(ani)
print(budi)

print("Ani tries to pick up at once: " + str(ani.pick_up()))
ani.start_washing()
ani.mark_ready()
print("After washing and drying:     " + ani.get_status())
print("Ani picks up now: " + str(ani.pick_up()))

print(ani)
print(budi)
```

To run the Java version from a terminal, both files must be compiled, because `OrderDemo` uses `Order`:

```
javac Order.java OrderDemo.java
java OrderDemo
```

(`javac *.java` compiles every Java file in the folder at once. In VS Code, simply click **Run** above `main` in `OrderDemo.java`; VS Code compiles everything the program needs.) For Python, `order.py` and `order_demo.py` must be in the same folder; then `python order_demo.py` runs it (`python3` on macOS and Linux). The Java version prints:

```
Order #1 for Ani: 3.5 kg, Rp 24500, RECEIVED
Order #2 for Budi: 5.0 kg express, Rp 52500, RECEIVED
Ani tries to pick up at once: false
After washing and drying:     READY
Ani picks up now: true
Order #1 for Ani: 3.5 kg, Rp 24500, PICKED UP
Order #2 for Budi: 5.0 kg express, Rp 52500, RECEIVED
```

and the Python version prints:

```
Order #1 for Ani: 3.5 kg, Rp 24500, RECEIVED
Order #2 for Budi: 5.0 kg express, Rp 52500, RECEIVED
Ani tries to pick up at once: False
After washing and drying:     READY
Ani picks up now: True
Order #1 for Ani: 3.5 kg, Rp 24500, PICKED UP
Order #2 for Budi: 5.0 kg express, Rp 52500, RECEIVED
```

The two outputs agree in every character but two: Java prints the true-or-false values as `true` and `false`, and Python as `True` and `False`, because that is how each language spells them.

Trace the Java version line by line, with **Figure 1.12** beside you.

- `Order ani = new Order(1, "Ani", 3.5, false);` does three things. `new Order(...)` creates a new object and runs the constructor with the four values, which sets its fields and its status to RECEIVED. `Order ani` declares a variable that can refer to an `Order`. And `=` makes `ani` refer to the new object. Python's `ani = Order(1, "Ani", 3.5, False)` does the same without the word `new` and without a type. Budi's order is created the same way, with express set to true.
- `System.out.println(ani);` prints the object; Java calls `ani.toString()` behind the scenes, which builds the first output line. Its price, 24500, was computed by the object itself by Equation (1.1); Budi's 52500 by Equation (1.2).
- `ani.pickUp()` sends the message *pickUp* to Ani's order while its status is RECEIVED. The method checks, finds the status is not READY, and returns `false` — the red loop of **Figure 1.12**. The output line says so, and the status is still RECEIVED.
- `ani.startWashing();` and `ani.markReady();` move the order along two black arrows. We ignore the `true` each returns.
- `ani.getStatus()` now reports `READY`, and `ani.pickUp()` succeeds and returns `true`.
- The last two lines print both orders again. Ani's is PICKED UP; Budi's is still RECEIVED, because every message was sent to `ani` and Budi's order has its own, separate state. That is **identity** at work: two objects of the same class, each minding its own business.

> **EXAMPLE.** Try to break the rule. Add the line `ani.status = "PICKED UP";` to `OrderDemo.java` before the first `println` and compile. The compiler refuses with `error: status has private access in Order`, and no program is produced at all. The mistake that **Listing 1.1** made silently is now impossible to make in Java without changing the `Order` class itself. Remove the line again before continuing.

1.8.6 What a variable of a class type holds

One more picture completes your mental model. When you write `Order ani = new Order(...)`, the variable `ani` does not contain the order itself; it contains a **reference** — in effect, the address where the object lives in the computer's memory. The object lives in a region of memory called the **heap**. You can draw it like this:

```
variables                     objects on the heap

ani  [ o-]--------------->  +---------------------------+
                            | Order                     |
                            | number    = 1             |
                            | customer  = "Ani"         |
                            | kilograms = 3.5           |
                            | express   = false         |
                            | status    = "PICKED UP"   |
                            +---------------------------+

budi [ o-]--------------->  +---------------------------+
                            | Order                     |
                            | number    = 2             |
                            | customer  = "Budi"        |
                            | kilograms = 5.0           |
                            | express   = true          |
                            | status    = "RECEIVED"    |
                            +---------------------------+
```

The distinction matters as soon as you write `Order x = ani;`. That line does not copy the order; it copies the *reference*, so that `x` and `ani` now point to the same object, and `x.pickUp()` changes the very order that `ani` sees. Python behaves the same way: every Python variable holds a reference to an object. C programmers will recognize references as a safer cousin of pointers — you can follow them, but you cannot do arithmetic on them or point them at arbitrary memory. Chapter 3 returns to references, and to what happens to an object when no variable refers to it any more.

1.8.7 What the class has bought us

Compare **Listing 1.9** with **Listing 1.1**. The ledger had the same information and the same price formula, but its rules lived nowhere. The class puts its one rule in exactly one place, the three status methods, and every program that uses orders gets that rule for free and cannot get around it. If Bu Sari adds a step, say IRONING between WASHING and READY, the change is made in one class and every program follows.

The class is not finished. It still accepts an order of minus five kilograms, because the constructor does not check its parameters; the status is a string, so a typing slip such as `"REDAY"` inside the class would go unnoticed; and prices are fixed in the code. Chapter 3 teaches the tools to check it, Chapter 5 teaches exceptions for rejecting bad input properly, and later chapters replace the status string with a type that cannot be misspelled. Good classes grow one careful step at a time.

1.9 The Road Map of This Book

This book is paced like a one-semester university course: sixteen chapters, one for each week. By its end you will be able to **explain** the concepts of object-oriented programming and the language features that support them; **analyze** a problem and **model** its solution with objects, as Section 1.3 began to do for the laundry shop; and **implement** that solution in two object-oriented languages — designing, building and testing programs of good quality, on your own and in a team. **Table 1.7** lays out the route.

Table 1.7 The sixteen chapters of this book and what each one adds.

Ch	Title	What it adds
1	Objects, Programs, and Your First Workbench	what programs and objects are; Java and Python; VS Code
2	The Development Environment: Editing, Running, and Debugging	Eclipse and VS Code; projects and packages; the debugger; packaging
3	Classes, Objects, Strings, Types, Conditionals, and Loops	the grammar of both languages; constructing objects; references
4	Evaluation and Discussion I	worked problems on Chapters 1–3
5	Arrays, Lists, Input, Exceptions, and Files	collections of objects; reading input; handling errors; files and streams
6	Documentation, Testing, and Interfaces	Javadoc and docstrings; JUnit and `unittest`; interfaces; overloading
7	Inheritance, Polymorphism, and Access Control	extending classes; overriding; abstract classes; casting; access levels
8	Evaluation and Discussion II	worked problems on Chapters 5–7
9	Collections and Generics	lists, sets and maps; generic types
10	Graphical User Interfaces and Event Handling	Swing and Tkinter; painting; events; inner classes and closures
11	The Model–View–Controller Pattern	a complete application built in three cooperating parts
12	Evaluation and Discussion III	worked problems on Chapters 9–11
13	Iterators, Immutability, and More Collections	walking through collections; objects that never change
14	Threads, Races, Deadlock, and Livelock	doing several things at once, safely
15	Sockets, Networking, and the Team Project	programs that talk over a network; building in a team
16	Evaluation and Discussion IV	worked problems on Chapters 13–15 and the whole book

Every fourth chapter — Chapters 4, 8, 12 and 16 — evaluates and discusses the three chapters before it: no new theory, but problems solved in full in both languages, with the reasoning shown step by step, and further problems for you to try.

The **Profitina Laundry** shop runs through the whole book, and each chapter adds to it: customers and a price list in Chapter 3, a file of the day's orders in Chapter 5, tests that prove the prices right in Chapter 6, express and dry-cleaning orders as subclasses in Chapter 7, a window for the counter in Chapter 10, several counters served at once in Chapter 14, and a shop that talks to its couriers over the network in Chapter 15. By the end, the three-line ticket of **Pseudocode 1.1** will have become a small but complete application.

Two books make good companions to this one. For Java, the long-standing textbook by Deitel and Deitel [24] covers the language in great depth, with many more exercises than any one book can hold. For Python, Lutz's *Learning Python* [25] explains the language's inner workings thoroughly. And for everything else, the official documentation of both languages, the Java Language Specification [3] and the Python Language Reference [8], is free on the web and more readable than its titles suggest.

1.10 Profitina in Practice

Profitina is an **AI visibility (GEO) platform**, live at `app.profitina.com`. GEO — Generative Engine Optimization — is the discipline of being *quotable by an answer engine*, as SEO was the

discipline of ranking in a list of links. The product scans a brand's website, measures how ready its content is to be cited by AI answer engines from real signals — whether the main content can be extracted cleanly, whether it carries structured data, whether the site is technically sound, down to a well-formed `robots.txt` — explains each signal it measured, and then helps fix what it found.

It is written in **Python**, as a FastAPI web application, and the ideas of this chapter are visible in its shape from the first file.

Signals are objects. A scan does not compute one enormous number in one enormous function. Each thing the scanner measures is a separate *signal*, an object of its own class in exactly the sense of Section 1.3: it holds its own state (what it found, the score it gave) and its own behavior (measure a page, score the finding, explain the score in plain words). The scanner behaves like the cashier of **Figure 1.3**: it does not know how any signal works inside; it sends each one the same messages — *measure this page*, *what is your score?*, *explain yourself* — and collects the answers. Adding a signal means writing one new class; the scanner does not change. That is the promise of abstraction from Section 1.3.5, and Chapter 6 makes it precise with an **interface** that every signal class implements.

The AI model sits behind an object. Profitina's Content Engine drafts GEO-friendly text with **Qwen2.5 running locally** on Profitina's own server through Ollama; no user data is sent to an outside AI service. The rest of the application never talks to the model directly. It talks to an object that wraps it, and that object has its own rules, just as `Order.pickUp()` has: when no model is available, it *says so* instead of silently returning a template, and a draft is served only when it scores better than a built-in template. Because those rules live in one class, no other part of the code can get around them by accident.

The law is part of the design. The platform handles personal data, so it is subject to Indonesia's Personal Data Protection Law, UU PDP 27/2022 [26]: consent must be explicit, data must be exportable, and an account purge must truly purge. Encapsulation is how a program keeps promises like these: if only one class may touch a user's personal data, there is exactly one place to check consent, one place that exports, one place that deletes. And one principle of the product applies to every class you will write in this book: **the app advises; the user decides.** A visibility score is a measurement, never a guarantee.

1.10.1 The worked case study: a day at Profitina Laundry

The hands-on case study of this book is always **Profitina Laundry**, a small teaching business, never the real product. Here is a whole day at its counter, built from the `Order` class of Section 1.8 without changing a line of it. **Listing 1.13** and **Listing 1.14** create four orders, move them through the day, and add up the cash from the orders that were collected.

Listing 1.13 `ShopDay.java`: a day at Profitina Laundry, using the unchanged `Order` class; compile it together with `Order.java`.

```java
public class ShopDay {
    public static void main(String[] args) {
        Order[] orders = {
            new Order(101, "Ani", 3.5, false),
            new Order(102, "Budi", 5.0, true),
            new Order(103, "Citra", 2.0, false),
            new Order(104, "Dewi", 7.5, true)
        };

        // The morning shift: every order goes into a machine.
        for (Order o : orders) {
            o.startWashing();
        }
        // Around noon, the first three come out dry and folded.
        orders[0].markReady();
        orders[1].markReady();
        orders[2].markReady();
        // In the afternoon, Ani and Citra collect theirs.
        orders[0].pickUp();
        orders[2].pickUp();
        // Dewi arrives too early and is politely refused.
        boolean dewiOk = orders[3].pickUp();

        long cash = 0;
        int waiting = 0;
        for (Order o : orders) {
            System.out.println(o);
            if (o.getStatus().equals("PICKED UP")) {
                cash = cash + o.price();
            } else {
                waiting = waiting + 1;
            }
        }
        System.out.println("Dewi's early pick-up accepted? " + dewiOk);
        System.out.println("Cash collected today: Rp " + cash);
        System.out.println("Orders still in the shop: " + waiting);
    }
}
```

Listing 1.14 `shop_day.py`: the same day in Python; it imports `Order` from `order.py` in the same folder.

```python
from order import Order

orders = [
    Order(101, "Ani", 3.5, False),
    Order(102, "Budi", 5.0, True),
    Order(103, "Citra", 2.0, False),
    Order(104, "Dewi", 7.5, True),
]

# The morning shift: every order goes into a machine.
for o in orders:
    o.start_washing()
# Around noon, the first three come out dry and folded.
orders[0].mark_ready()
orders[1].mark_ready()
orders[2].mark_ready()
# In the afternoon, Ani and Citra collect theirs.
orders[0].pick_up()
orders[2].pick_up()
# Dewi arrives too early and is politely refused.
dewi_ok = orders[3].pick_up()

cash = 0
waiting = 0
for o in orders:
    print(o)
    if o.get_status() == "PICKED UP":
        cash = cash + o.price()
    else:
        waiting = waiting + 1
print("Dewi's early pick-up accepted? " + str(dewi_ok))
print("Cash collected today: Rp " + str(cash))
print("Orders still in the shop: " + str(waiting))
```

Compile and run the Java version with `javac Order.java ShopDay.java` and `java ShopDay`, the Python version with `python shop_day.py`. The Java version prints:

```
Order #101 for Ani: 3.5 kg, Rp 24500, PICKED UP
Order #102 for Budi: 5.0 kg express, Rp 52500, READY
Order #103 for Citra: 2.0 kg, Rp 14000, PICKED UP
Order #104 for Dewi: 7.5 kg express, Rp 78750, WASHING
Dewi's early pick-up accepted? false
Cash collected today: Rp 38500
Orders still in the shop: 2
```

and the Python version prints the same lines, except that its fifth line reads `Dewi's early pick-up accepted? False`.

Two new pieces of grammar appear. `Order[] orders = { ... };` in Java, and `orders = [ ... ]` in Python, make a collection of four orders; Chapter 5 treats arrays and lists fully. And `for (Order o : orders)` in Java, `for o in orders:` in Python, visit each order in turn, calling it `o` — a loop that reads almost like English.

Check the numbers. Dewi's 7.5 kilograms express costs $p(7.5) = 52500$ plus $\lfloor 52500/2 \rfloor = 26250$, that is 78750, by Equation (1.2). The cash is Ani's 24500 plus Citra's 14000, that is 38500; Budi's order is ready but not collected, and Dewi's is still washing, so two orders remain in the shop. And notice the line that matters most: Dewi came too early, and the order *refused*. `ShopDay` did not need a single line of its own to enforce that rule. The class did it.

1.11 Common Mistakes, and How to Avoid Them

Running an old version of a Java program. You edit `Hello.java`, run `java Hello`, and see the old output. You forgot to compile again; `java` ran the old `Hello.class`. Cure: after every edit, run `javac` first — or use the Run button in VS Code, which compiles for you, or `java Hello.java` for a single file.

File name and class name that differ. A file `hello.java` containing `public class Hello` is rejected: *class Hello is public, should be declared in a file named Hello.java*. The rule includes capital letters, even on Windows, where the file system itself does not care. Cure: name the file exactly after its public class.

Forgetting that Java is case-sensitive. `system.out.println` with a small s is rejected with the puzzling message *package system does not exist*, because Java takes `system` to be the name of a package. `System`, `String` and `Math` are capitalized; `main`, `println` and `equals` are not. Python is equally case-sensitive: `Print("hi")` fails with *NameError: name 'Print' is not defined*. Cure: copy names exactly as the book writes them, and let VS Code's completion suggest them.

Missing semicolons in Java, stray indentation in Python. Each language has one piece of punctuation that beginners miss. Java needs `;` at the end of each statement, and its message *';' expected* points just after the place where it is missing. Python needs consistent indentation; a line pushed right for no reason gives *IndentationError: unexpected indent*. Cure: read the line the error names, and the line *before* it.

Comparing strings with `==` in Java. In Java, `==` between two objects asks whether they are *the same object* (the same reference, in the picture of Section 1.8.6), not whether they contain the same characters. Two strings with the same text can be different objects, so `status == "READY"` may be false even when the status reads READY. Cure: always compare strings in Java with `equals`, as **Listing 1.9** does. In Python, `==` compares contents and is correct.

Making fields public "to save time". It saves a minute today and costs a day next month, when a line far away writes nonsense into the field, exactly as in **Listing 1.1**. Cure: make every field `private` (in Python, give it a leading underscore), and add a method for each thing outsiders legitimately need to do.

Confusing a class with an object. "I changed `Order`'s status" is a sentence that makes no sense: the class has no status; each object has its own. Cure: when you speak about state, name the object — "Ani's order", `ani` — not the class.

Fighting the installation instead of reading the message. Nearly every setup failure — a tool that is "not recognized", the wrong version, the wrong Python in VS Code, a terminal that was open before the installation — is listed in **Table 1.6**. Cure: read the first line of the message, find it in the table, and apply the fix before trying anything else.

Formatting numbers with the computer's regional settings. Java's `String.format("%.1f", 3.5)` prints `3.5` on a computer set to English but `3,5` on one set to Indonesian, because it follows the regional convention for decimal commas. That is why the listings of this chapter join numbers with `+` instead, which always prints `3.5`. Cure: when output must be the same everywhere — in files, in tests, in this book — either avoid regional formatting or name the convention explicitly; Chapter 5 shows how.

Believing a program because it runs. A program that prints *something* is not thereby correct. The ledger of **Listing 1.1** ran perfectly and printed a negative price. Cure: check every output by

hand against a formula, as this chapter did with Equation (1.1) and Equation (1.2); Chapter 6 turns such checks into automatic tests.

1.12 Chapter Summary

A **program** is an algorithm — a finite list of precise steps from an input to an output — written in a programming language. Java is **compiled** by `javac` into **bytecode**, which the **Java Virtual Machine** runs on any platform; Python is run by an **interpreter** that compiles to bytecode internally each time. In Java you compile and then run; in Python you just run.

Object-oriented programming organizes a program as a society of **objects**, each with **state** (what it knows), **behavior** (what it can do) and **identity** (it is itself and no other). A **class** is the blank form; each object is an **instance** with its own values. Objects cooperate by sending **messages**, written with a dot: `ani.price()`. **Abstraction** keeps only what matters and hides *how* behind *what*. Encapsulation, abstraction, inheritance and polymorphism are taught in full in Chapters 3, 6 and 7.

Parallel arrays and free functions put no one in charge of keeping data sensible; any line can corrupt it. An object bundles its data with the only methods allowed to change it, and those methods can **refuse** a request that makes no sense.

Java (1995, Sun Microsystems, now Oracle) is portable, widely supported and explicit about types and access; Python (1991) is concise, dynamically typed and dominant in data and AI work; both are in the top three of every major popularity measurement. This book uses **Java 25 LTS** (Eclipse Temurin) and **Python 3.14**, and every program also runs on Java 17+ and Python 3.10+.

The workbench is **VS Code** with the **Extension Pack for Java** and the **Python** extension. Check the tools with `java -version`, `javac -version` and `python --version` (`python3` on macOS and Linux); run with the Run button or from the terminal; debug with a breakpoint, `F5`, `F10`, `F11` and the Variables panel.

The first class, `Order`, has five **private** fields, a **constructor**, a `price()` method implementing Equation (1.1) and Equation (1.2), and three status methods that move the order along **Figure 1.12** and refuse to move it at the wrong moment. A variable of a class type holds a **reference** to an object on the heap, not the object itself.

1.13 Exercises

1.1 Choose a place you know well that is not a laundry shop — a school canteen, a *warung*, a bus terminal, a library. Name four kinds of objects in it, and for each give at least three items of state and three behaviors, in the form of **Table 1.1**.

1.2 Sort the following into classes and objects, and for each object say which class it is an instance of: `WashingMachine`; the machine by the door, called Machine A; `Customer`; Budi; `Order`; order number 104; Machine B.

1.3 Using Equation (1.1) and Equation (1.2), compute by hand the price of (a) 2.3 kilograms regular, (b) 7.5 kilograms express, (c) 0.9 kilograms express. Then check (b) against the output of **Listing 1.13**.

1.4 Using Equation (1.3) and Equation (1.4), compute the number of loads for orders of 16, 17 and 1 kilograms, first with machines of capacity $c = 8$, then with $c = 7$. Show the whole-number division for each.

1.5 Change **Listing 1.3** and **Listing 1.4** so that each prints a third line, `Open every day from 07.00 to 20.00`. Run both. Which line did you add in each language, and where exactly must it go?

1.6 Predict what happens in each case, then try it: (a) **Listing 1.3** is saved as `hello.java` with a small h; (b) the semicolon at the end of the first `println` line of **Listing 1.3** is deleted; (c) `System` is written `system`; (d) the second line of a Python file is indented by four spaces while the first is not.

1.7 Suppose the two lines `ani.startWashing();` and `ani.markReady();` of **Listing 1.11** were swapped, so that `ani.markReady()` is called *before* `ani.startWashing()`. What would each of those two calls return, and what would `ani.getStatus()` report afterward? Use **Figure 1.12** to justify your answer.

1.8 In **Listing 1.13**, add the line `orders[1].pickUp();` just after Citra's pick-up. What are the new cash total and the new number of orders still in the shop? Check by running it.

1.9 Write a class `WashingMachine`, in Java and in Python, following the description of Section 1.3.1: a machine has a name, a capacity in whole kilograms, the kilograms currently loaded, and whether it is running. It offers `load(kg)`, which refuses (returns false) if the machine is running, if `kg` is not positive, or if the load would exceed the capacity; `start()`, which refuses if the machine is already running or empty; `finish()`, which stops the machine, empties it and returns the kilograms washed (or 0 if it was not running); and a one-line description. Give it a `main` that exercises every refusal.

1.10 The field `status` of `Order` is `private`, and the fields `number`, `customer`, `kilograms` and `express` are also `final`. Explain in one or two sentences each: (a) what could go wrong if `status` were public; (b) why `status` cannot be `final`; (c) what `final` protects against for `kilograms`.

1.11 Bu Sari wants every order to record the customer's telephone number. List every change needed in the procedural **Listing 1.1**, and every change needed in the object-oriented **Listing 1.9** and **Listing 1.11**. Which version needs changes in more places, and why?

1.12 (a) In Python, what does `9 / 8` evaluate to, and what does `9 // 8`? What would **Listing 1.6** print for the second order if its line used `/` instead of `//` and the result were then converted to a whole number with `int(...)`? (b) Show that Equation (1.4) holds for $c = 1$ and every whole number $w \geq 0$.

1.14 Solutions to the Exercises

Solution 1.1. For a school canteen, one good answer is the following. A *food stall* knows its name, its menu with prices, and whether it is open; it can take an order, report a price, and open or close. A *student* knows a name, a class, and the money in pocket; he or she can order, pay and eat. A *tray* knows which dishes it carries and whether it has been returned; it can be loaded, carried and returned. A *cashier's box* knows today's cash and the list of payments; it can receive a payment, give change and report the day's total. The important thing in any answer is the split: state is information, in nouns; behavior is action, in verbs; and each behavior uses or changes that object's own state.

Solution 1.2. The classes are `WashingMachine`, `Customer` and `Order` — each is a kind of thing, a blank form. The objects are Machine A and Machine B, both instances of `WashingMachine`; Budi, an instance of `Customer`; and order number 104, an instance of `Order`. A useful test: a class name answers "what kind?", an object answers "which one?".

Solution 1.3. (a) $p(2.3) = \text{round}(7000 \times 2.3) = \text{round}(16100) = 16100$, so Rp 16,100. (In the computer, 7000×2.3 comes out as 16099.999999999998 because 2.3 cannot be stored exactly in binary; the rounding in Equation (1.1) is what makes the result exactly 16100.) (b) $p(7.5) = 52500$ and $p_{\text{express}}(7.5) = 52500 + \lfloor 52500/2 \rfloor = 52500 + 26250 = 78750$, so Rp 78,750 — exactly what **Listing 1.13** printed for Dewi. (c) $p(0.9) = 6300$ and $p_{\text{express}}(0.9) = 6300 + 3150 = 9450$, so Rp 9,450.

Solution 1.4. With $c = 8$, Equation (1.4) computes $\lfloor (w + 7)/8 \rfloor$. For $w = 16$: $\lfloor 23/8 \rfloor = 2$ (since $23 = 2 \times 8 + 7$). For $w = 17$: $\lfloor 24/8 \rfloor = 3$. For $w = 1$: $\lfloor 8/8 \rfloor = 1$. These agree with Equation (1.3): $\lceil 2 \rceil = 2$, $\lceil 2.125 \rceil = 3$, $\lceil 0.125 \rceil = 1$. With $c = 7$ the formula is $\lfloor (w + 6)/7 \rfloor$. For $w = 16$: $\lfloor 22/7 \rfloor = 3$ (since $22 = 3 \times 7 + 1$), and indeed $\lceil 16/7 \rceil = \lceil 2.29 \rceil = 3$. For $w = 17$: $\lfloor 23/7 \rfloor = 3$, and $\lceil 2.43 \rceil = 3$. For $w = 1$: $\lfloor 7/7 \rfloor = 1$.

Solution 1.5. In Java, add `System.out.println("Open every day from 07.00 to 20.00");` as a new line *inside* `main`, after the second `println` and before the closing brace of `main`; indent it like its neighbors. Outside `main` it would not compile, because Java statements must live inside a method. In Python, add `print("Open every day from 07.00 to 20.00")` as a new last line, *not* indented. Both programs then print their two earlier lines followed by `Open every day from 07.00 to 20.00`.

Solution 1.6. (a) The compiler refuses, and on the book's machine printed exactly:

```
hello.java:1: error: class Hello is public, should be declared in a file named Hello.java
public class Hello {
       ^
1 error
```

(b) The compiler reports the missing semicolon and points just after where it belongs:

```
Hello.java:3: error: ';' expected
        System.out.println("Hello from Profitina Laundry!")
                                                          ^
1 error
```

(c) The compiler thinks `system` is the name of a package and prints `Hello.java:3: error: package system does not exist`, with a caret under the dot after `system`. (d) Python refuses before running a single line. The last line of its message is `IndentationError: unexpected indent`, and the line above it shows the offending line; the first line of the message names the file and says `line 2`. In every case the lesson is the same: read the file name and line number in the message, look at that line and the one before it, and fix the first error first.

Solution 1.7. `ani.markReady()` would return `false`: `markReady()` succeeds only from WASHING, and the order is still RECEIVED, so the method refuses and changes nothing. Then `ani.startWashing()` would return `true` and move the order to WASHING. `ani.getStatus()` would report `WASHING`, not READY. In **Figure 1.12**, the only arrow leaving RECEIVED is `startWashing()`; any other message sent there is refused. (Consequently the later `ani.pickUp()` in **Listing 1.11** would also return `false`.)

Solution 1.8. Budi's order is READY at that point, so `orders[1].pickUp()` succeeds. The cash becomes $24500 + 52500 + 14000 = 91000$, and only Dewi's order remains, so the program

prints `Cash collected today: Rp 91000` and `Orders still in the shop: 1`, and Budi's line ends in `PICKED UP`. Running the modified program confirms both numbers.

Solution 1.9. **Listing 1.15** and **Listing 1.16** are one solution. Like `Order`, each method checks first and acts only if the request makes sense.

Listing 1.15 `WashingMachine.java`: a washing machine that refuses overloads, loading while running, and starting while empty.

```
public class WashingMachine {
    private final String name;
    private final int capacity;     // kilograms
    private int loadKg;             // kilograms inside right now
    private boolean running;

    public WashingMachine(String name, int capacity) {
        this.name = name;
        this.capacity = capacity;
        this.loadKg = 0;
        this.running = false;
    }

    public boolean load(int kg) {
        if (running || kg <= 0 || loadKg + kg > capacity) {
            return false;           // refuse: running, nonsense, or too heavy
        }
        loadKg = loadKg + kg;
        return true;
    }

    public boolean start() {
        if (running || loadKg == 0) {
            return false;           // nothing to wash, or already washing
        }
        running = true;
        return true;
    }

    public int finish() {
        if (!running) {
            return 0;
        }
        int washed = loadKg;
        running = false;
        loadKg = 0;
        return washed;              // kilograms that came out clean
    }

    @Override
    public String toString() {
        return name + " [" + loadKg + "/" + capacity + " kg, "
                + (running ? "running" : "idle") + "]";
    }
```

Listing 1.15 (Continued) *WashingMachine.java: a washing machine that refuses overloads, loading while running, and starting while empty.*

```
    public static void main(String[] args) {
        WashingMachine m = new WashingMachine("Machine A", 8);
        System.out.println(m);
        System.out.println("load 5 kg: " + m.load(5));
        System.out.println("load 4 kg: " + m.load(4));
        System.out.println("load 3 kg: " + m.load(3));
        System.out.println("start:     " + m.start());
        System.out.println(m);
        System.out.println("load 1 kg: " + m.load(1));
        System.out.println("finish:    " + m.finish() + " kg washed");
        System.out.println(m);
    }
}
```

Listing 1.16 `washing_machine.py`: the same class in Python; the block under `if __name__ == "__main__":` runs only when the file is run directly, not when it is imported.

```
class WashingMachine:
    def __init__(self, name, capacity):
        self._name = name
        self._capacity = capacity     # kilograms
        self._load_kg = 0             # kilograms inside right now
        self._running = False

    def load(self, kg):
        if self._running or kg <= 0 or self._load_kg + kg > self._capacity:
            return False              # refuse: running, nonsense, or too heavy
        self._load_kg = self._load_kg + kg
        return True

    def start(self):
        if self._running or self._load_kg == 0:
            return False              # nothing to wash, or already washing
        self._running = True
        return True

    def finish(self):
        if not self._running:
            return 0
        washed = self._load_kg
        self._running = False
        self._load_kg = 0
        return washed                 # kilograms that came out clean

    def __str__(self):
        return (self._name + " [" + str(self._load_kg) + "/" + str(self._capacity)
                + " kg, " + ("running" if self._running else "idle") + "]")


if __name__ == "__main__":
    m = WashingMachine("Machine A", 8)
    print(m)
    print("load 5 kg: " + str(m.load(5)))
    print("load 4 kg: " + str(m.load(4)))
    print("load 3 kg: " + str(m.load(3)))
    print("start:     " + str(m.start()))
    print(m)
    print("load 1 kg: " + str(m.load(1)))
    print("finish:    " + str(m.finish()) + " kg washed")
    print(m)
```

The Java version prints:

```
Machine A [0/8 kg, idle]
load 5 kg: true
load 4 kg: false
load 3 kg: true
start:     true
Machine A [8/8 kg, running]
load 1 kg: false
finish:    8 kg washed
Machine A [0/8 kg, idle]
```

and the Python version prints the same with `True` and `False` capitalized. Trace the refusals: 5 kilograms fit into an empty 8-kilogram machine; 4 more would make $9 > 8$ and are refused; 3 more make exactly 8 and are accepted; after `start()`, loading even 1 kilogram is refused because the machine is running; `finish()` returns the 8 kilograms washed and leaves the machine empty and idle. A complete answer also tests `start()` on an empty machine and `load(0)`; both return false.

Solution 1.10. (a) If `status` were public, any line of any program could write any text into it — `"PICKED UP"` for laundry still in the machine, or a misspelling such as `"REDAY"` — and the forward-only rule of **Figure 1.12** would be as unprotected as in **Listing 1.1**. (b) `final` means "assigned once and never again", but the whole purpose of `status` is to change as the laundry moves through the shop, so it cannot be `final`. (c) `final` on `kilograms` guarantees that even the class's *own* methods cannot change the weight after the constructor has set it, so the price computed at the counter can never silently change later.

Solution 1.11. In the procedural version: add a fifth array `phone` in `main`, with one entry per order and in the same positions as the others; add a `phone` parameter to `printAll` and pass the new array at *both* calls; and change the printing line. Every future function that deals with orders must also receive the fifth array. In the object-oriented version: add one `private final String phone;` field and one constructor parameter to `Order`, assign it in the constructor, and add it to `toString()` if it should be printed; in `OrderDemo`, add the telephone number to the two `new Order(...)` calls. Methods such as `price()` and `pickUp()` do not change at all. The procedural version needs changes wherever orders are *passed around*; the object version needs changes only where orders are *defined* and *created*, because an order travels as one object and carries its new field with it.

Solution 1.12. (a) `9 / 8` is `1.125`: in Python, `/` always produces a fractional number. `9 // 8` is `1`: whole-number division, which discards the remainder. `int(9 / 8)` is also `1`, because `int` cuts off the fraction. So a version of **Listing 1.6** using `int(orders[i] / capacity)` would print `order 2: 9 kg -> 1 load(s)` — wrong, since 9 kilograms cannot fit into one 8-kilogram machine — and the day's total would be too small. The correct expression is `(orders[i] + capacity - 1) // capacity`, which gives 2. (b) For $c = 1$ the right-hand side of Equation (1.4) is $\lfloor (w + 1 - 1)/1 \rfloor = \lfloor w \rfloor = w$, because w is a whole number, and the left-hand side is $\lceil w/1 \rceil = \lceil w \rceil = w$ for the same reason. The two sides are equal for every whole $w \geq 0$. ■

References

[1] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, and K. A. Houston, *Object-Oriented Analysis and Design with Applications*, 3rd ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2007.

[2] A. C. Kay, "The early history of Smalltalk," *ACM SIGPLAN Notices*, vol. 28, no. 3, pp. 69–95, Mar. 1993.

[3] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 25 Edition*. Redwood City, CA, USA: Oracle America, 2025. [Online]. Available: https://docs.oracle.com/javase/specs/

[4] Oracle, "Oracle Java SE support roadmap." Accessed: Sep. 23, 2026. [Online]. Available: https://www.oracle.com/java/technologies/java-se-support-roadmap.html

[5] Oracle, "The arrival of Java 27!," *Inside.java*, Sep. 15, 2026. [Online]. Available: https://inside.java/2026/09/15/jdk-27-available/

[6] G. van Rossum, "A brief timeline of Python," *The History of Python*, Jan. 2009. [Online]. Available: https://python-history.blogspot.com/2009/01/brief-timeline-of-python.html

[7] H. van Kemenade, "PEP 745 – Python 3.14 release schedule," Python Enhancement Proposals, 2024. [Online]. Available: https://peps.python.org/pep-0745/

[8] Python Software Foundation, *The Python Language Reference*, release 3.14, 2025. [Online]. Available: https://docs.python.org/3.14/reference/

[9] Python Software Foundation, "Python 3.14.7," Aug. 5, 2026. [Online]. Available: https://www.python.org/downloads/latest/python3.14/

[10] Stack Overflow, "2025 Developer Survey: Technology — most popular technologies, programming, scripting, and markup languages." Accessed: Sep. 24, 2026. [Online]. Available: https://survey.stackoverflow.co/2025/technology

[11] P. Carbonnelle, "PYPL PopularitY of Programming Language index." Accessed: Sep. 24, 2026. [Online]. Available: https://pypl.github.io/PYPL.html

[12] StatisticsTimes, "Top computer languages 2026" (PYPL index, worldwide, April 2026). Accessed: Sep. 24, 2026. [Online]. Available: https://statisticstimes.com/tech/top-computer-languages.php

[13] TIOBE Software, "TIOBE index for September 2026." Accessed: Sep. 24, 2026. [Online]. Available: https://www.tiobe.com/tiobe-index/

[14] TechRepublic, "TIOBE Index for September 2026: top 10 most popular programming languages," Sep. 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://www.techrepublic.com/article/news-tiobe-index-language-rankings/

[15] GitHub, "Octoverse: A new developer joins GitHub every second as AI leads TypeScript to #1," *The GitHub Blog*, Oct. 2025. Accessed: Sep. 24, 2026. [Online]. Available: https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/

[16] Eclipse Adoptium, "Eclipse Temurin." Accessed: Sep. 23, 2026. [Online]. Available: https://adoptium.net/

[17] Eclipse Foundation, "Eclipse IDE." Accessed: Sep. 23, 2026. [Online]. Available: https://www.eclipse.org/

[18] Apache Software Foundation, "Apache NetBeans." Accessed: Sep. 23, 2026. [Online]. Available: https://netbeans.apache.org/

[19] Microsoft, "Getting started with Java in VS Code." Accessed: Sep. 23, 2026. [Online]. Available: https://code.visualstudio.com/docs/java/java-tutorial

[20] Microsoft, "Getting started with Python in VS Code." Accessed: Sep. 23, 2026. [Online]. Available: https://code.visualstudio.com/docs/python/python-tutorial

[21] Eclipse Adoptium, "Eclipse Temurin 8u502, 11.0.32, 17.0.20, 21.0.12, 25.0.4 and 26.0.2 available," Aug. 2026. [Online]. Available: https://adoptium.net/news/2026/08/eclipse-temurin-8u502-11032-17020-21012-2504-2602-available

[22] J. Gibbons, "JEP 330: Launch single-file source-code programs," OpenJDK, 2018. [Online]. Available: https://openjdk.org/jeps/330

[23] R. Pressler, J. Gibbons, and G. Bierman, "JEP 512: Compact source files and instance main methods," OpenJDK, 2025. [Online]. Available: https://openjdk.org/jeps/512

[24] P. Deitel and H. Deitel, *Java How to Program, Early Objects*, 10th ed. Boston, MA, USA: Pearson, 2015.

[25] M. Lutz, *Learning Python*, 5th ed. Sebastopol, CA, USA: O'Reilly Media, 2013.

[26] Republic of Indonesia, *Law Number 27 of 2022 on Personal Data Protection* (UU PDP), Oct. 2022.

Chapter 2

The Development Environment: Editing, Running, and Debugging

By the end of this chapter you will be able to explain, in plain words, what an integrated development environment is and what it does for you; say exactly what happens between the moment you press *Run* and the moment your program prints its first line, both for Java (compiler, bytecode, Java Virtual Machine) and for Python (interpreter, bytecode, Python virtual machine); set up and find your way around two professional workbenches, Eclipse and Visual Studio Code, side by side; organize code into projects, packages and modules and connect them with imports; use content assist, quick fixes and refactoring so that the tool writes the boring parts for you; and, above all, *debug*: stop a running program at a line of your choosing, step through it one statement at a time, watch its variables change, read its call stack, set a breakpoint that fires only under a condition you write, and read a stack trace from the top line to the bottom. You will do all of this on a real bug in a real program, the price calculator of Profitina Laundry, in Java and in Python, and you will see the bug found step by step rather than guessed at.

2.1 Where We Left Off

Chapter 1 put a workbench on your desk. You installed a Java Development Kit (Eclipse Temurin, Java 25) and Python 3.14, you set up Visual Studio Code with the Extension Pack for Java and the Python extension, you ran your first programs in both languages, and in Section 1.7.7 you watched a loop run in the debugger for the first time. You also wrote your first class, a laundry `Order`, and met the central idea of this book: a program is a community of *objects*, each holding its own data and offering its own behavior.

That chapter moved quickly on purpose. This one slows down and looks at the workbench itself, because you will spend hundreds of hours at it. A carpenter who does not know which chisel does what can still build a table, but slowly and with a lot of splinters.

Three questions organize the chapter.

1. **What actually happens when I run a program?** Section 2.3 answers with *compiling*, *interpreting* and the *virtual machine*, so that every error message has a place in the story.
2. **How do I drive my tools well?** Sections 2.4 to 2.9 walk through Eclipse and VS Code side by side. The ideas are the same in both; only the buttons move.
3. **When the program is wrong, how do I find out why?** Sections 2.10 to 2.13 teach debugging from zero and use it on a laundry bill that is too high, a day's takings that are too low, and a program that crashes.

> **KEY INSIGHT.** You will not remember every menu path in this chapter, and you do not need to. What you need to keep is the *vocabulary* (workspace, project, package, perspective, breakpoint, step over, step into, call stack) and the *habit* of reaching for the debugger instead of guessing. Menus can be looked up; habits cannot.

2.2 What an IDE Is, and Why You Want One

2.2.1 A workbench, not a magic box

You can write a Java or Python program with nothing more than a plain text editor such as Notepad, and then compile and run it from a terminal. Many experienced programmers still do this for small scripts, and in Section 2.3 we will do it ourselves on purpose, so that you can see what the tools do behind the curtain.

For anything larger than a few dozen lines, though, the plain-editor approach becomes painful. You must remember every method name in every file, type long commands to compile and run, count to line 47 when the compiler complains about line 47, and sprinkle print statements through the code whenever it prints a wrong number.

An **integrated development environment**, or **IDE**, puts all of these activities in one window and connects them to one another. *Integrated* is the important word. The editor knows about the compiler, so it underlines your mistakes while you type. The compiler knows about the editor, so clicking an error message jumps to the line. The debugger knows about both, so when a program stops at a breakpoint the editor highlights the exact line and shows the values of the variables beside it.

> **DEFINITION.** An *integrated development environment* (IDE) is a program that combines, in one window, a source-code editor, a way to build (compile) and run programs, a debugger, and tools to navigate and restructure the code, all sharing the same understanding of the project.

An IDE is not magic. Everything it does, you could do by hand with separate tools, and when it surprises you, you should be able to ask "which underlying tool did this, and what would I type to do it myself?" Section 2.3 answers that for running programs, and Section 2.11 for debugging.

2.2.2 The five jobs of an IDE

It helps to name the jobs an IDE does, because every menu in Eclipse and every panel in VS Code belongs to one of them. **Table 2.1** lists them.

Table 2.1 The five jobs every IDE does, and where this chapter covers each.

Job	What it means	Where you meet it in this chapter
Edit	Type and change source code, with color, indentation, completion and instant error marks	Sections 2.5, 2.6 and 2.9
Build	Turn source code into something the machine can run (compile, package into a JAR)	Sections 2.3 and 2.6
Run	Start the program with the right settings and show its output	Sections 2.6 and 2.7
Debug	Pause the program, step through it, inspect and change its state	Sections 2.10 to 2.13
Navigate and restructure	Find where something is defined, rename it everywhere, organize imports	Sections 2.8 and 2.9

2.2.3 Eclipse and VS Code: two workbenches, one set of ideas

This book uses two tools, and it is worth saying why.

Eclipse is a free, open-source IDE maintained by the Eclipse Foundation [1]. It runs on Windows, macOS and Linux and has been used for more than twenty years to develop Java applications of

every kind, including (for many years) Android apps. It is a *full* IDE with deep built-in knowledge of Java [2].

Visual Studio Code (VS Code) is a free, open-source *editor* from Microsoft that becomes an IDE for a language when you install that language's extensions [3], [4]. With the Extension Pack for Java it understands Java projects; with the Python extension it understands Python. It is lighter than Eclipse, it treats Java and Python the same way, and it is what Chapter 1 installed for you.

Because this book teaches every idea in Java *and* in Python, you will use VS Code for both languages most of the time. We still teach Eclipse carefully: its vocabulary (workspace, perspective, view, Package Explorer) is the classic vocabulary of Java tooling that other IDEs such as IntelliJ IDEA borrow, and it makes visible some ideas that VS Code hides, such as the difference between a *view* and an *editor*.

Table 2.2 is the dictionary you will keep coming back to. Do not try to memorize it now; each row is explained in the sections that follow.

Table 2.2 The same ideas under two names: a dictionary between Eclipse and VS Code.

Idea	Eclipse	VS Code
Where your settings and projects live	Workspace (a folder chosen at start-up)	The folder you open (*File > Open Folder*), optionally a multi-root `.code-workspace` file
A unit of code that is built together	Java project (with a `.project` file)	The opened folder; Java projects are detected automatically and shown in the *Java Projects* view
Browsing the code	Package Explorer or Project Explorer view	Explorer view and Java Projects view
Arrangement of panels for a task	Perspective (Java, Debug, ...)	One layout; the Run and Debug view appears when debugging
Which Java or Python runs the program	Installed JREs (*Window > Preferences > Java > Installed JREs*)	Java runtime set in *Java: Configure Java Runtime*; Python interpreter chosen with *Python: Select Interpreter*
Saved way of running a program	Launch configuration (*Run > Run Configurations...*)	`launch.json` in the `.vscode` folder
Error list	Problems view	Problems panel
Completion while typing	Content Assist (Ctrl+Space)	IntelliSense (Ctrl+Space)
Fix a marked error	Quick Fix (Ctrl+1)	Quick Fix (Ctrl+.)

2.3 How a Program Runs: Compiling, Interpreting, and Virtual Machines

Section 1.2.3 sketched how source code becomes a running program. Before we open either IDE, we look closer, because without this picture the IDE is a box with a green button, and with it every message the IDE shows you has a place in the story.

2.3.1 The processor speaks only one language

The processor inside your computer, the *CPU*, understands exactly one kind of instruction: **machine code**, a sequence of numbers each meaning something very small, such as "add these two registers" or "jump to this address". Different processor families speak different machine codes: *x86-64* for Intel and AMD, *ARM64* for recent Macs and most phones, and a program for one does not run on the other.

Nobody writes programs in machine code any more. We write in *high-level languages* such as Java and Python, which read almost like English and mathematics:

```
total = total + line
```

Something must translate that line into machine code before the CPU can do anything with it. There are two classic ways to translate, and both Java and Python use a mixture of them.

2.3.2 Compilers and interpreters, and when each complains

Section 1.2.3 compared a **compiler** to a translator who translates a whole book once and prints it, and an **interpreter** to a translator who stands beside you and translates aloud as you go. The comparison has a consequence that matters every day: the book translator finds a nonsensical sentence *before* anyone reads the book, while the live translator finds it only when the reader reaches it, perhaps halfway through.

> **DEFINITION.** A *compiler* translates a whole program into another form before it runs. An *interpreter* carries out a program's instructions one after another while it runs. A *compile-time error* is found by the compiler before the program starts; a *run-time error* happens while the program is running.

Real languages mix the two ideas. Both Java and Python first *compile* your source into an intermediate language called **bytecode**, and then a **virtual machine** *interprets* the bytecode. The difference is *when* each step happens and who does it, and **Figure 2.1** shows it at a glance.

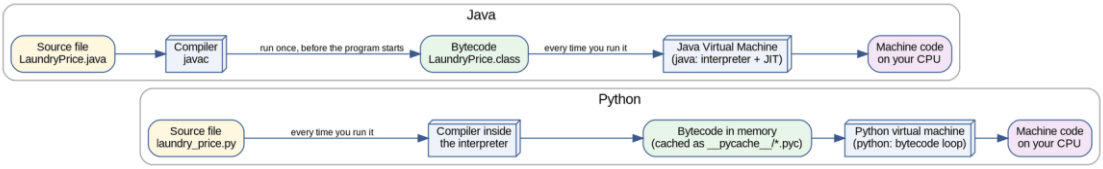


Figure 2.1 From source code to the processor. Java compiles once with javac and the JVM runs the bytecode every time; Python compiles inside the interpreter every time the program starts, then its virtual machine runs the bytecode.

2.3.3 Java: javac, bytecode, and the Java Virtual Machine

In Java, translation happens in two separate steps performed by two separate programs, both of which come with the Java Development Kit (JDK).

Step 1: compile with `javac`. The Java compiler, `javac`, reads your `.java` file, checks that it follows every rule of the Java language (every statement ends with a semicolon, every variable has a declared type, every method you call exists), and writes a `.class` file. The `.class` file does not contain machine code for your CPU. It contains **Java bytecode**, the instruction set of an imaginary computer called the **Java Virtual Machine** (JVM), defined precisely in the JVM specification [5].

Step 2: run with `java`. The `java` launcher starts a JVM, which loads the `.class` file and executes its bytecode. The JVM is itself a real program, written for your real CPU. It first *interprets* the bytecode, one instruction at a time. While doing so it counts how often each method runs; methods that run many times ("hot" methods) are handed to a **just-in-time compiler** (JIT) that translates them into real machine code for your CPU, so that they run at full speed from then on.

Why two steps? Because the same `.class` file runs on every machine that has a JVM, Windows or Linux, Intel or ARM: "write once, run anywhere".

Let us watch both steps on the smallest interesting program we can write. **Listing 2.1** defines a method that adds two numbers and a `main` method that uses it.

Listing 2.1 `Add.java`: a two-line method, small enough to read its bytecode.

```
public class Add {

    static int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        System.out.println("2 + 3 = " + add(2, 3));
    }
}
```

To compile and run it in a terminal (on Windows use *Command Prompt* or *PowerShell*; on macOS and Linux use *Terminal*), go to the folder that holds the file and type:

```
javac Add.java
java Add
```

The first command produces `Add.class` and prints nothing if all is well. The second prints:

```
2 + 3 = 5
```

Now for the part that is usually hidden. The JDK includes a disassembler, `javap`, that prints the bytecode inside a `.class` file in readable form [6]. Typing `javap -c Add` shows:

```
Compiled from "Add.java"
public class Add {
  public Add();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  static int add(int, int);
    Code:
       0: iload_0
       1: iload_1
       2: iadd
       3: ireturn

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #7                  // Field
java/lang/System.out:Ljava/io/PrintStream;
       3: iconst_2
       4: iconst_3
       5: invokestatic  #13                 // Method add:(II)I
       8: invokedynamic #19,  0             // InvokeDynamic
#0:makeConcatWithConstants:(I)Ljava/lang/String;
      13: invokevirtual #23                 // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
      16: return
}
```

Read the method `add` first. Its four bytecode instructions say: `iload_0`, push the first `int` parameter (`a`) onto a small working stack; `iload_1`, push the second (`b`); `iadd`, pop both, add them, push the sum; `ireturn`, return the `int` on top of the stack. That is exactly `return a + b;`, spelled for a machine that works with a stack of values. In `main`, `iconst_2` and `iconst_3` push the constants 2 and 3, `invokestatic` calls `add`, and the remaining instructions build the string `"2 + 3`

`= 5"` and call `println`. You also see a method `Add()` that you never wrote: it is the *default constructor* that the compiler adds to every class without one, and Chapter 3 explains it.

You will rarely read bytecode, but one fact matters: **the JVM runs not your `.java` file but the `.class` file that `javac` made from it.** Editing the source and running again without recompiling runs the *old* program; an IDE hides this by recompiling every time you save, a terminal does not.

> **WARNING.** Since Java 11 you may also type `java Add.java` (with the `.java` extension). This *source-file mode* compiles the file in memory and runs it at once, which is handy for single-file programs such as most listings in this book. It does not leave a `.class` file behind, and it works only when the whole program is in that one file.

2.3.4 Python: the interpreter, its hidden compiler, and `__pycache__`

In Python the two steps happen inside one program, the Python interpreter, which you start by typing `python` (on Windows, often `py`) or `python3` (on macOS and Linux). When you type

```
python3 add_bytecode.py
```

the interpreter reads the whole file, *compiles* it to Python bytecode in memory, and then its virtual machine, a loop inside the interpreter that fetches and executes one bytecode instruction after another, runs that bytecode [7]. There is no separate compiler command, and there is (usually) no bytecode file left on the disk.

Python lets you look at its bytecode from inside a program, with the standard module `dis` (short for *disassembler*). **Listing 2.2** is the twin of **Listing 2.1**: it runs a tiny `add` function and then disassembles it.

Listing 2.2 `add_bytecode.py`: running a function, then printing the bytecode Python made for it.

```
"""add_bytecode.py: runs a tiny function, then shows the bytecode Python made for it."""
import dis


def add(a, b):
    return a + b


print("2 + 3 =", add(2, 3))
dis.dis(add)
```

On Python 3.11 the output is:

```
2 + 3 = 5
  5           0 RESUME                   0

  6           2 LOAD_FAST                0 (a)
              4 LOAD_FAST                1 (b)
              6 BINARY_OP                0 (+)
             10 RETURN_VALUE
```

The numbers 5 and 6 are source line numbers. `RESUME` marks the start of the function, the two `LOAD_FAST` instructions push `a` and `b`, `BINARY_OP 0 (+)` adds them, and `RETURN_VALUE` returns the sum. The two virtual machines are remarkably alike: both are *stack machines* that load, add and return. The difference is that the JVM has one instruction per type (`iadd` for `int`s, `dadd` for `double`s), while Python's single `BINARY_OP +` decides at run time what "plus" means, because Python variables have no declared types.

> **WARNING.** The exact bytecode differs between Python versions; Python 3.12, 3.13 and 3.14 print slightly different instruction names and offsets from the 3.11 output shown here, and

> some add colored or adaptive forms. The idea (load, load, add, return) is the same in all of them. Bytecode is an internal detail of the interpreter, not part of the language.

Where is the bytecode kept? For the file you run directly, only in memory. But when your program *imports* another module of yours (Section 2.8), Python saves that module's bytecode in a folder called `__pycache__` next to the source, so that the next start-up can skip the compile step if the source has not changed. You can see this for yourself with the file of **Listing 2.4** (Section 2.6): importing it from a one-line command and then listing the folder shows

```
python3 -c "import my_first_module"
```

```
.
./__pycache__
./__pycache__/my_first_module.cpython-311.pyc
./my_first_module.py
```

The name `cpython-311` records the interpreter (CPython, version 3.11) that made it. You never need to open or copy `__pycache__` folders; they are safe to delete.

2.3.5 When mistakes are caught: compile time and run time

Now we can explain something that confuses almost every beginner who learns both languages: **Java refuses to start a program that Python would happily start and then crash.**

Take the first program of Section 2.6 and delete one semicolon, so that line 6 reads `System.out.println("Hello Eclipse!")`. Compiling it gives:

```
src/myfirstproject/MyFirstClass.java:6: error: ';' expected
        System.out.println("Hello Eclipse!")
                                           ^
1 error
```

Read the message from left to right: the *file* (`src/myfirstproject/MyFirstClass.java`), the *line number* (6), the *kind* of message (`error`), and the *explanation* (`';' expected`). The caret `^` on the third line points to the exact column where the compiler expected the semicolon. No `.class` file is written, so there is nothing to run: the error was caught at *compile time*.

Python also compiles before running, so a *syntax* error, a line that does not follow the grammar of the language, is also caught before anything runs. Remove the colon after `def main()` in **Listing 2.4** and run it:

```
  File "/home/student/hello_py/my_first_module.py", line 4
    def main()
              ^
SyntaxError: expected ':'
```

Again you get the file, the line (4), a caret under the place where the colon should be, and the kind of error (`SyntaxError: expected ':'`). Nothing was printed, because the program never started.

The difference appears with mistakes that are grammatical but *meaningless*, such as a name that does not exist. Java's compiler checks every name and type before the program runs; Python's checks only the grammar and looks names up as each line runs. Section 2.8.4 shows both reactions to a forgotten `import`: a compile-time `cannot find symbol` in Java, a run-time `NameError` in Python. A misspelled name in a rarely used Python branch can therefore hide for months, one reason why testing (Chapter 6) matters even more in Python.

> **KEY INSIGHT.** *Java checks names and types before running; Python checks them while running.* Java's strictness costs you a few more lines of declarations and buys you earlier error

> messages. Python's flexibility saves typing and moves some errors to run time. Neither is better in every situation; knowing which one you are in tells you where to look for a mistake.

2.4 Installing and Starting Eclipse

2.4.1 Downloading and installing

Eclipse is downloaded from the Eclipse Foundation's download page, `https://www.eclipse.org/downloads/` [1]. The page offers the **Eclipse Installer**, a small program that downloads and sets up the particular *package* of Eclipse you choose. On Windows the installer is an `.exe` file; on macOS it is delivered as a packaged application that you install the usual Mac way; on Linux it is a compressed archive that you unzip first and then start from the unzipped folder.

When the installer starts, it shows a list of Eclipse packages, each bundling the plug-ins for one kind of work. Choose **Eclipse IDE for Java Developers**, the one at the top in **Figure 2.2**. It contains the Java editor and compiler, a Git client, and support for the Maven and Gradle build tools, which is everything this book needs.

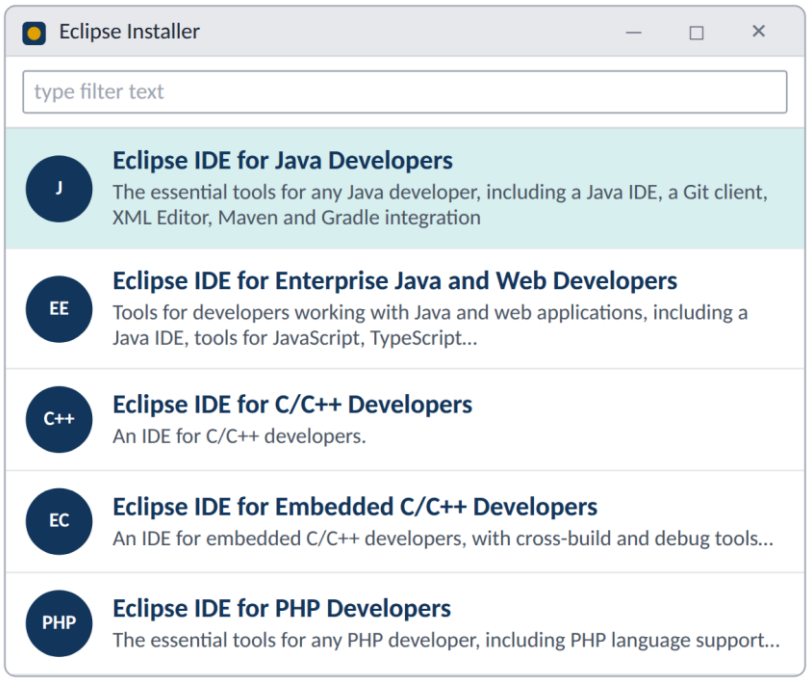


Figure 2.2 The Eclipse Installer's list of packages. Eclipse IDE for Java Developers is the one to choose.

The next screen, **Figure 2.3**, asks two questions. *Java 21+ VM* is the Java runtime that Eclipse itself will run on; the installer offers to download one, and the default is fine. *Installation Folder* is the parent folder in which an `eclipse` folder will be created; any folder you can write to will do, for example `D:\Program` on Windows or your home folder on macOS and Linux. Keep the two check boxes that create a start-menu entry and a desktop shortcut, click **INSTALL**, accept the license, and wait until the button turns into **LAUNCH**.

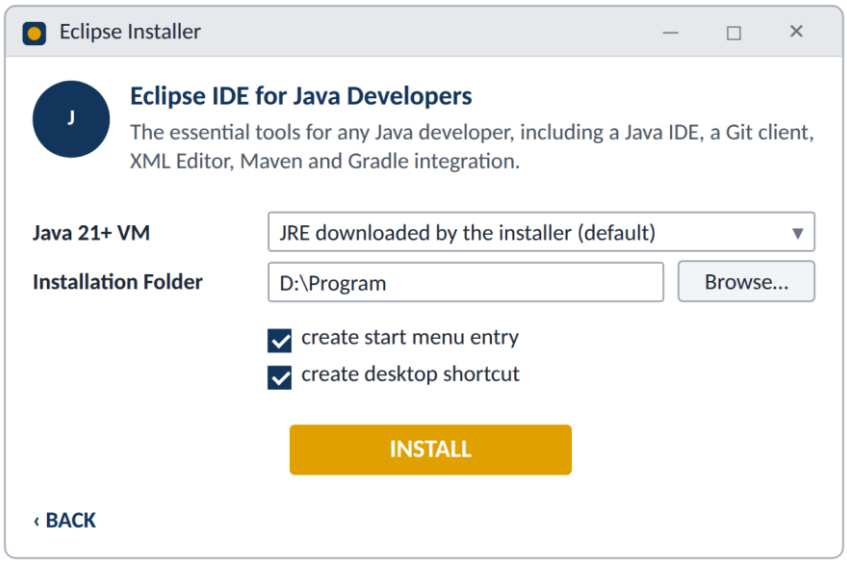


Figure 2.3 Choosing the Java runtime for Eclipse and the installation folder, just before pressing INSTALL.

> **WARNING.** The Java that *Eclipse runs on* and the Java that *your programs are compiled for* are two separate settings. Eclipse may run on the runtime it downloaded, while your projects use the JDK 25 you installed in Chapter 1. You set the latter under *Window > Preferences > Java > Installed JREs*; if the JDK you installed is missing from that list, click *Add...* and point it at the JDK folder.

2.4.2 The workspace

The very first thing Eclipse asks when it starts is where your **workspace** should be (**Figure 2.4**).

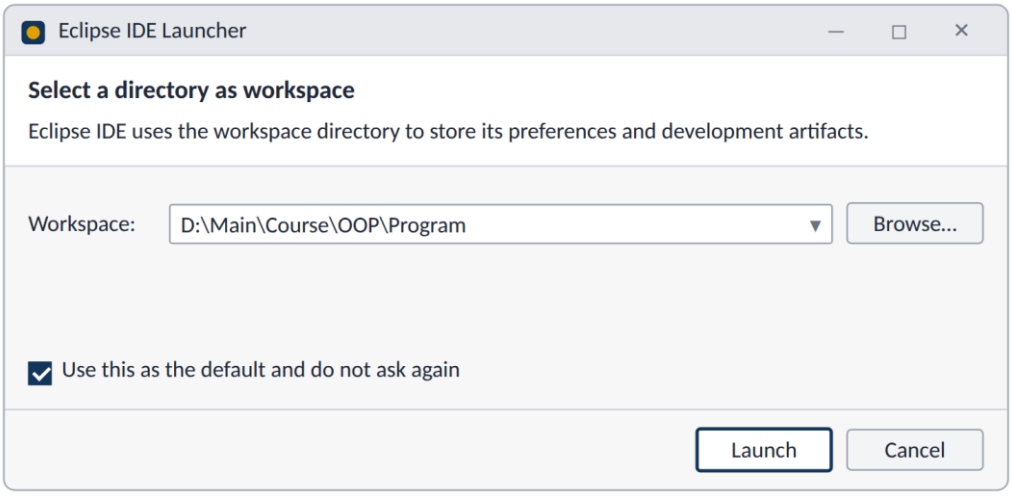


Figure 2.4 The Eclipse IDE Launcher asks for a workspace folder before the workbench opens.

> **DEFINITION.** An Eclipse *workspace* is a folder on your disk in which Eclipse stores its *metadata*, the preferences you set, the state of each plug-in, logs and the list of your projects, and in which it creates new projects by default.

The metadata lives in a hidden subfolder, `.metadata`. Your projects may live inside the workspace folder or anywhere else on the disk; a project kept in its own Git repository, for example, usually lives outside it.

Choose an empty folder, such as `D:\Main\Course\OOP\Program` on Windows or `~/oop/workspace` on macOS and Linux, tick *Use this as the default and do not ask again* if you like, and click **Launch**.

File > Switch Workspace > Other… (**Figure 2.5**) moves to another workspace later; one workspace per course keeps settings and experiments apart.

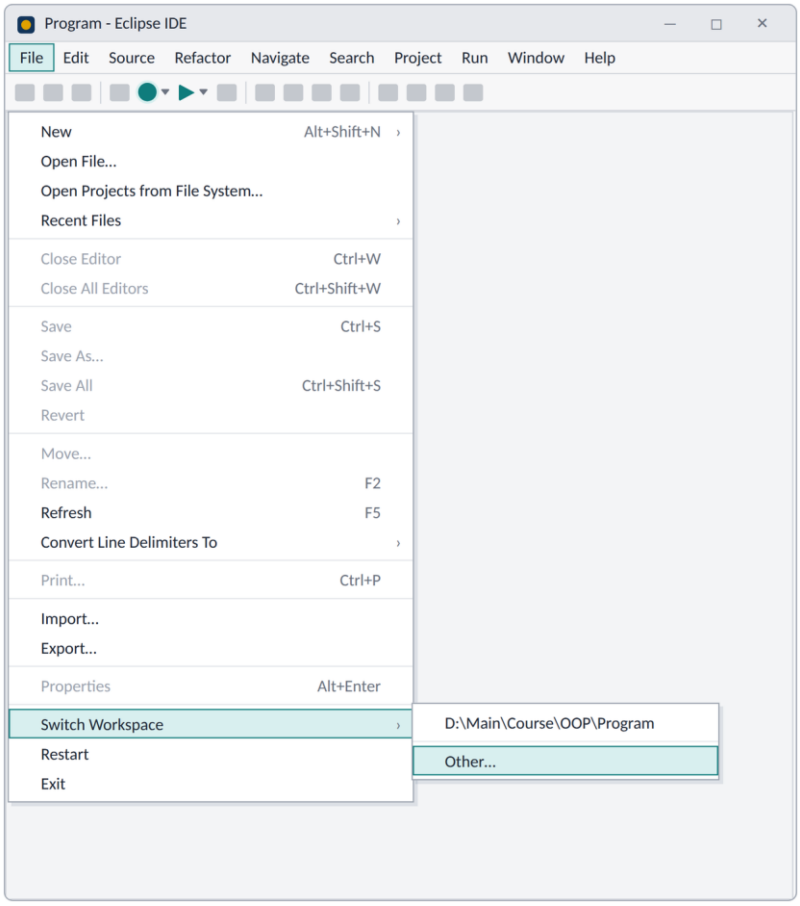


Figure 2.5 File > Switch Workspace > Other… opens the launcher again so that you can choose a different workspace.

2.4.3 Appearance

By default Eclipse uses a light color theme. If you prefer a dark or classic look, open *Window > Preferences > General > Appearance* (**Figure 2.6**), choose a theme, click *Apply and Close*, and restart Eclipse, because some of the operating system's own styling is applied only at start-up. On macOS the *Preferences* item is under the *Eclipse* menu instead of *Window*; this is the only place where the menu paths in this chapter differ between systems.

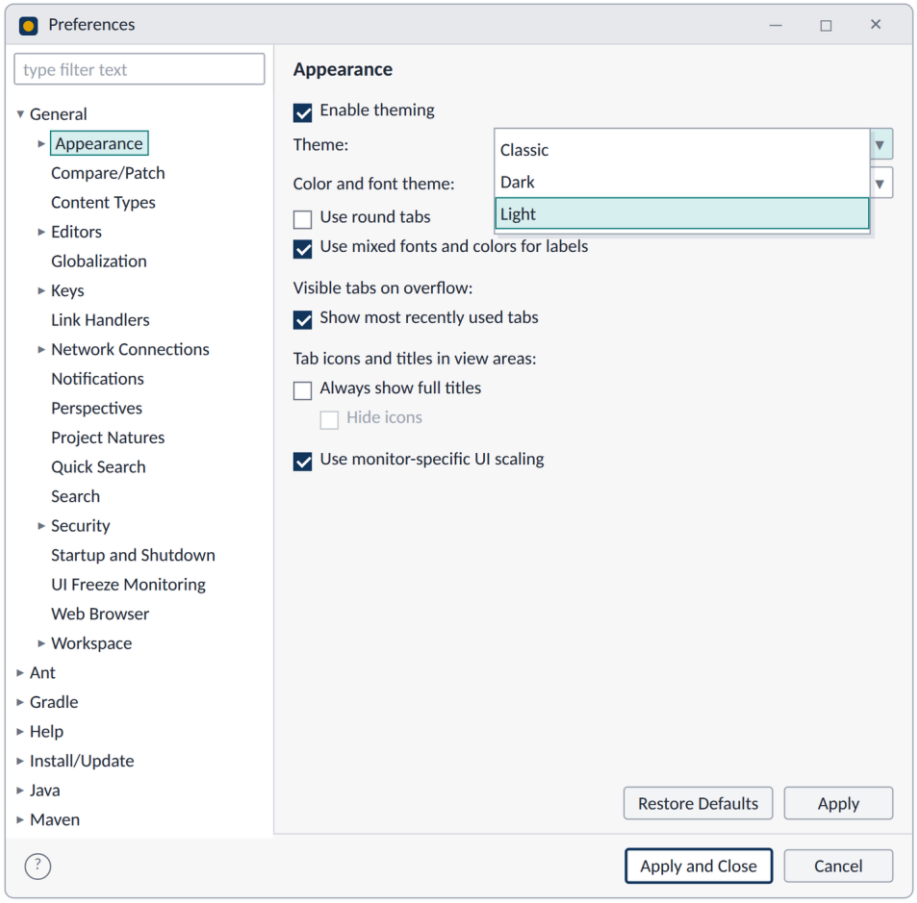


Figure 2.6 The Appearance preferences page, where the color theme is chosen.

2.5 The Eclipse Workbench: Views, Editors, and Perspectives

When the workspace opens, you see the **workbench**, the main Eclipse window. It is built from two kinds of parts, views and editors, arranged by a third idea, the perspective. These three words explain almost everything you see.

2.5.1 Views and editors

> **DEFINITION.** A *view* is a panel that shows structured information and usually lets you act on it directly, such as the tree of your projects or the list of errors. An *editor* is a panel that shows the contents of one file for you to change; changes in an editor are not written to disk until you save.

The difference matters in practice. If you rename a file in the Package Explorer *view*, the file is renamed on the disk immediately; there is nothing to save. If you type in an *editor*, the file on the disk is unchanged until you press Ctrl+S (Cmd+S on macOS). An editor with unsaved changes is called a *dirty* editor, and Eclipse marks it with an asterisk before the file name on its tab, for example `*MyFirstClass.java`. Learn to glance at the tabs for asterisks: running a program while its editor is dirty runs the last *saved* version, and "but I changed that line!" is one of the most common beginner complaints.

Several editors can be open at once, stacked in the middle of the window as tabs; drag a tab to see two files side by side.

2.5.2 Perspectives

> **DEFINITION.** A *perspective* is a named arrangement of views and editors suited to one task, such as writing Java code (the Java perspective) or debugging (the Debug perspective).

You will use two perspectives almost all the time: **Java** and **Debug**. Eclipse switches to the Debug perspective by itself when a program stops at a breakpoint (Section 2.10.3), and you can switch by hand with *Window > Perspective > Open Perspective*, shown in **Figure 2.7**. The *Other...* item opens the full list of perspectives available in your installation (**Figure 2.8**); which ones appear depends on the plug-ins installed.

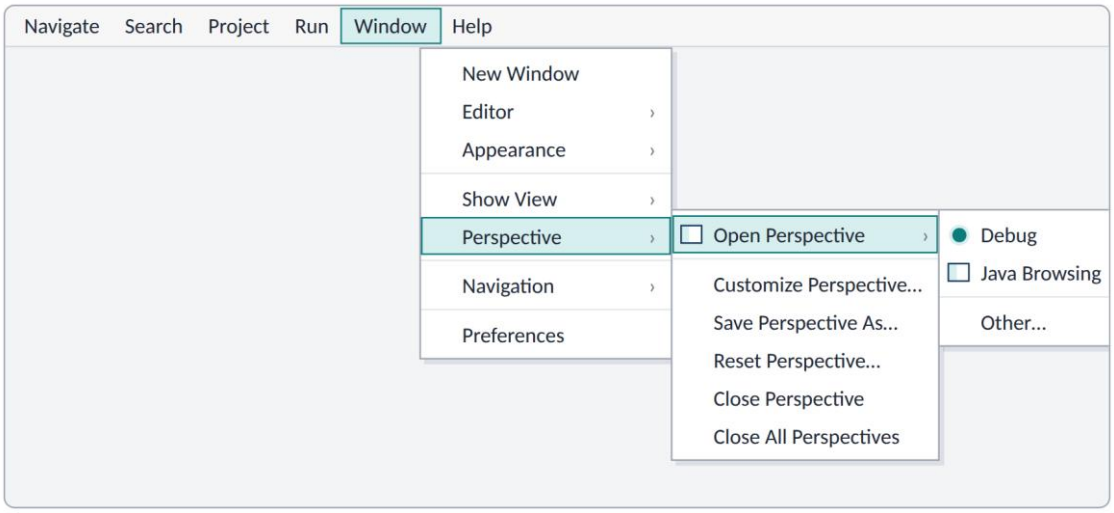


Figure 2.7 Window > Perspective > Open Perspective, with the Debug and Java Browsing perspectives offered directly and the rest under Other….

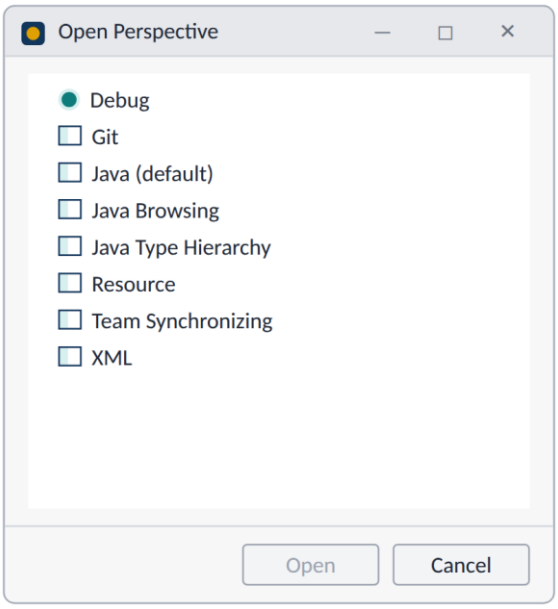


Figure 2.8 The Open Perspective dialog lists every perspective of the installation; Java is marked as the default.

Editors are shared between perspectives: if you have `Tester.java` open in the Java perspective and switch to the Debug perspective, the same editor is still there. Only the views around it change.

Figure 2.9 shows the Java perspective of a fresh workspace. On the left is the **Package Explorer**, the tree of your projects. The large empty area in the middle is the editor area. On the right is the

Outline view, which will show the structure (fields and methods) of whatever file is open. Along the bottom are the **Problems**, **Javadoc** and **Declaration** views. Because the workspace is still empty, the Package Explorer offers links to create a Java project or to import one.

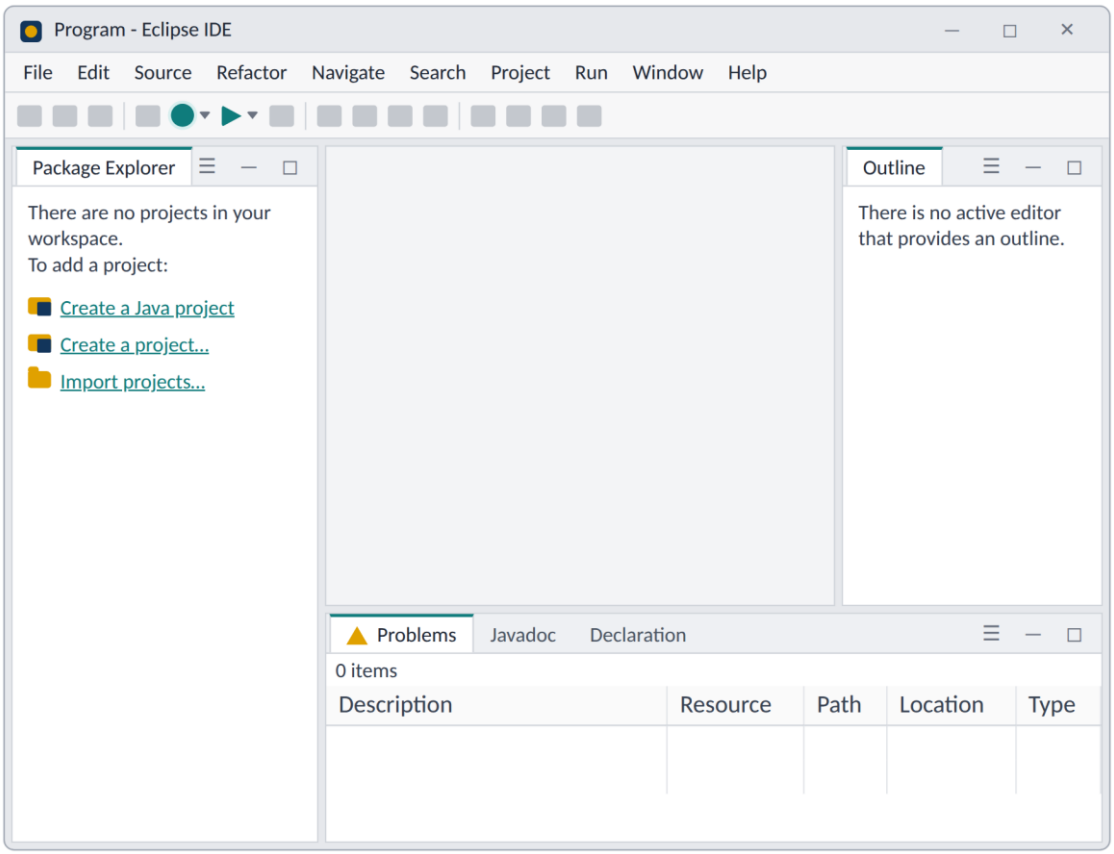


Figure 2.9 The Java perspective of an empty workspace: Package Explorer on the left, editor area in the middle, Outline on the right, Problems, Javadoc and Declaration along the bottom.

When you close a view by accident or the window stops making sense, *Window > Perspective > Reset Perspective...* restores the original arrangement, and *Window > Show View > Other...* (**Figure 2.10**) finds any view by name. *Save Perspective As...* and *Customize Perspective...* save your own arrangement and hide or show toolbar buttons.

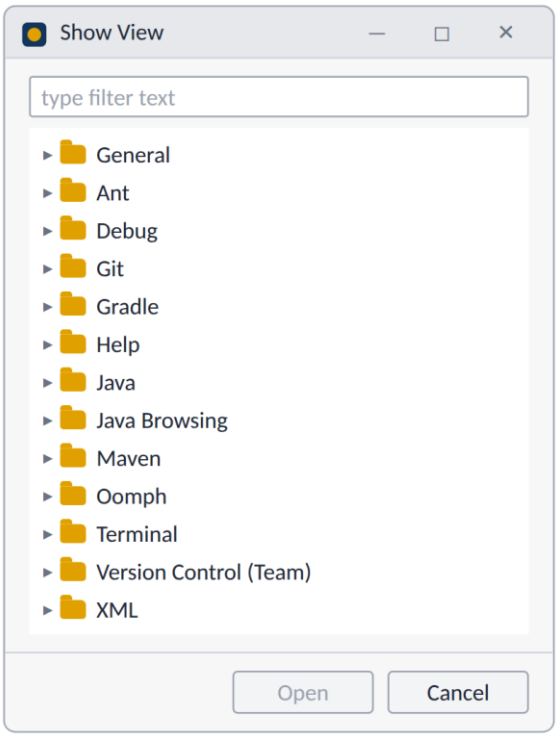


Figure 2.10 The Show View dialog, grouped by category; type part of a view's name in the filter box to find it.

2.5.3 The views of the Java perspective

Package Explorer. Browse your projects, open a file by double-clicking it, and change the structure (rename, delete, create packages and classes, drag files); a right-click shows what is possible. The *Project Explorer* shows the same projects as plain folders.

Outline. Shows the classes, fields and methods of the file in the active editor; click a method to jump to it.

Problems. The Problems view lists every error and warning in your projects, each with a description, the file, the folder and the line. Double-clicking a problem opens the file at that line. Its drop-down menu lets you filter what is shown, for example only the problems of the selected project, and a right-click on a problem offers **Quick Fix** (Ctrl+1), the same automatic repairs that we meet in the editor in Section 2.9. **Figure 2.11** shows Quick Fix offered for a warning about the project's Java version.

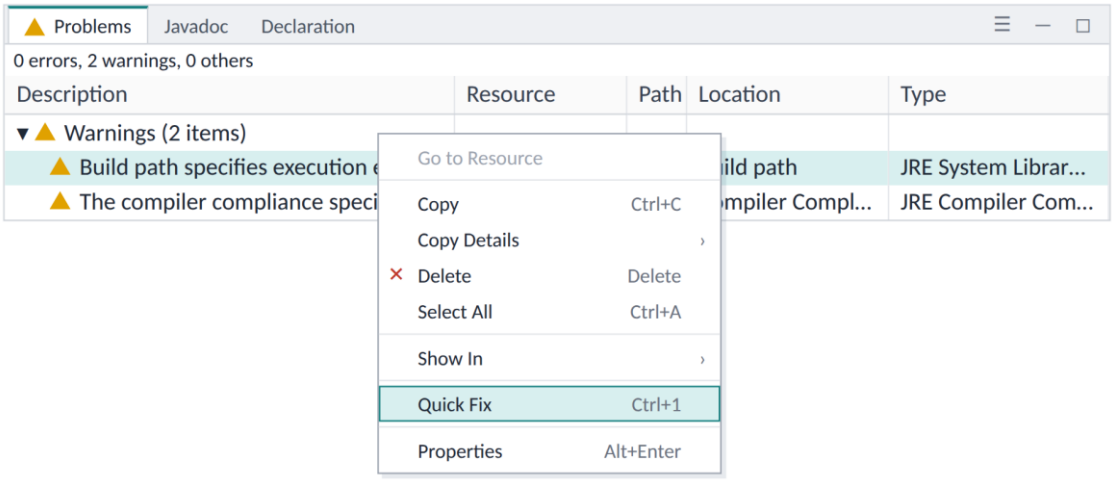


Figure 2.11 The Problems view with two warnings; the context menu offers Quick Fix.

Javadoc. The Javadoc view shows the documentation of whatever Java element the cursor is on. With the cursor on `println`, for example, it shows the official description of `PrintStream.println(String)` (**Figure 2.12**). Chapter 6 teaches you to write such documentation for your own classes.

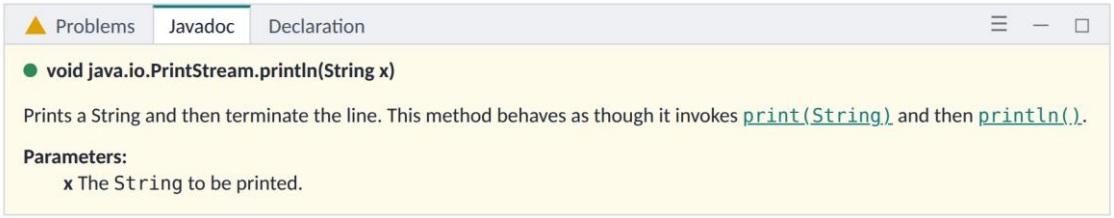


Figure 2.12 The Javadoc view showing the documentation of println(String) while the cursor is on a call to it.

The Java editor. Each Java file opens in its own Java editor. A right-click in the narrow gray strip to the left of the text, called the *ruler* or *gutter*, offers settings for the editor (**Figure 2.13**). Turn on **Show Line Numbers** now: every error message and every debugger report in this chapter refers to line numbers, and counting lines by eye is a waste of your life. The same menu contains **Toggle Breakpoint**, which we use from Section 2.10 onward.

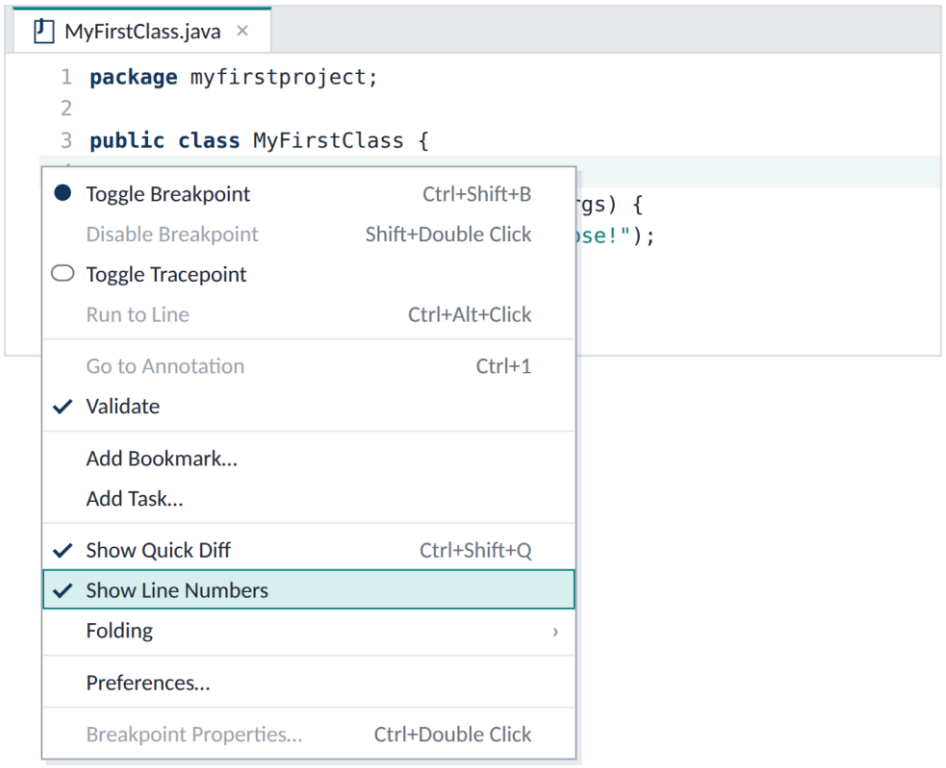


Figure 2.13 A right-click in the editor's left ruler: Show Line Numbers is ticked, and Toggle Breakpoint sits at the top.

2.5.4 Projects and their natures

An **Eclipse project** is a folder that groups the source files, configuration files and compiled files of one piece of work into a unit that can be built and reused. Each project has a small file called `.project` in its folder, and that file records the project's **natures**: labels that say what kind of project it is. A project with the *Java nature* is a Java project, so Eclipse compiles its `.java` files and offers Java tools for it. A project can have several natures at once, for example Java and Maven, to describe several technical aspects of the same code. You will rarely edit `.project` by hand, but knowing it exists explains why copying a folder into the workspace is not the same as importing a project (Section 2.8.6).

2.6 Your First Eclipse Project, Package, and Class

We now build, run and export the smallest possible Java program in Eclipse, and its Python twin in VS Code.

2.6.1 Creating the project

Select *File* > *New* > *Java Project*. In the dialog of **Figure 2.14**, type `MyFirstProject` as the project name. Leave *Use default location* ticked, so the project goes into the workspace. Under *JRE*, choose the Java version you want (Java 25 if you installed it in Chapter 1). Under *Project layout*, keep *Create separate folders for sources and class files*: this puts your `.java` files in a folder called `src` and the compiled `.class` files in a separate folder called `bin`, so the two never get mixed up. If a *Module* section offers to create a `module-info.java` file, leave that box unticked; modules are an advanced feature this book does not need. Press **Finish**.

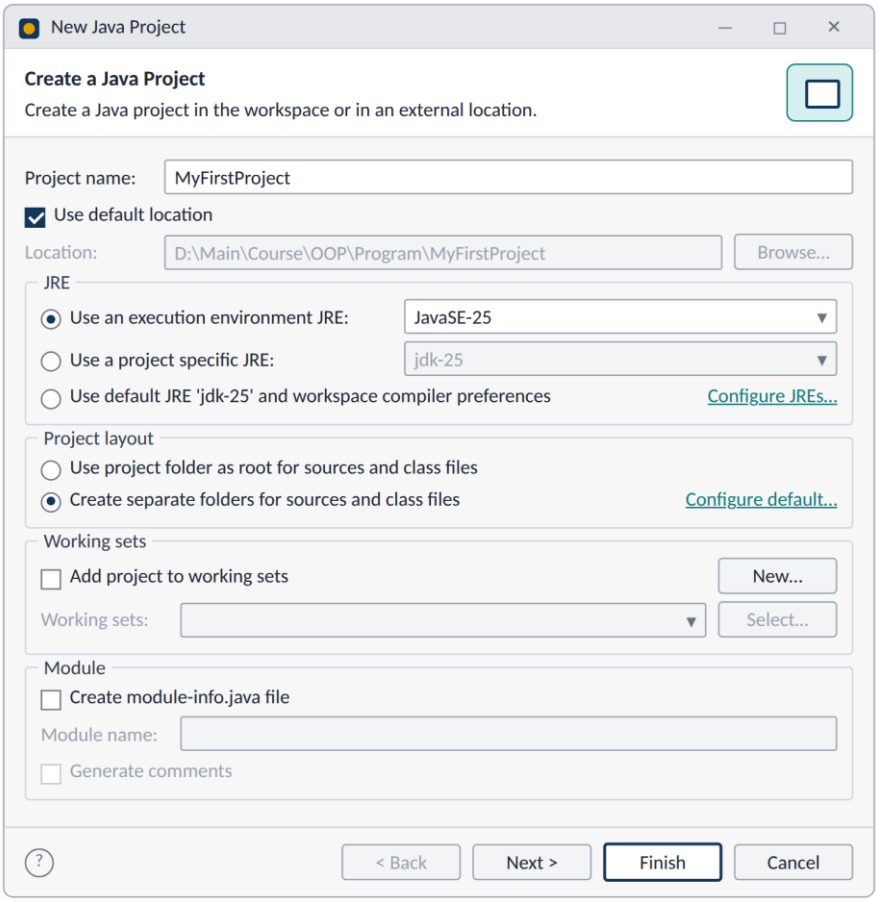


Figure 2.14 The New Java Project wizard, with separate folders for sources and class files.

The new project appears in the Package Explorer as a folder. Open it: you will see the JRE System Library (the standard Java classes your code can use) and an empty `src` folder.

2.6.2 Creating a package

A **package** is a named group of related classes; Section 2.8 explains packages fully. By convention, package names are written entirely in lowercase, and the top-level package of a project often has the same name as the project, which makes it easy to find the project that a piece of code belongs to. We therefore use the package name `myfirstproject`.

Right-click the `src` folder and choose *New* > *Package* (**Figure 2.15**). Type the name and press **Finish** (**Figure 2.16**).

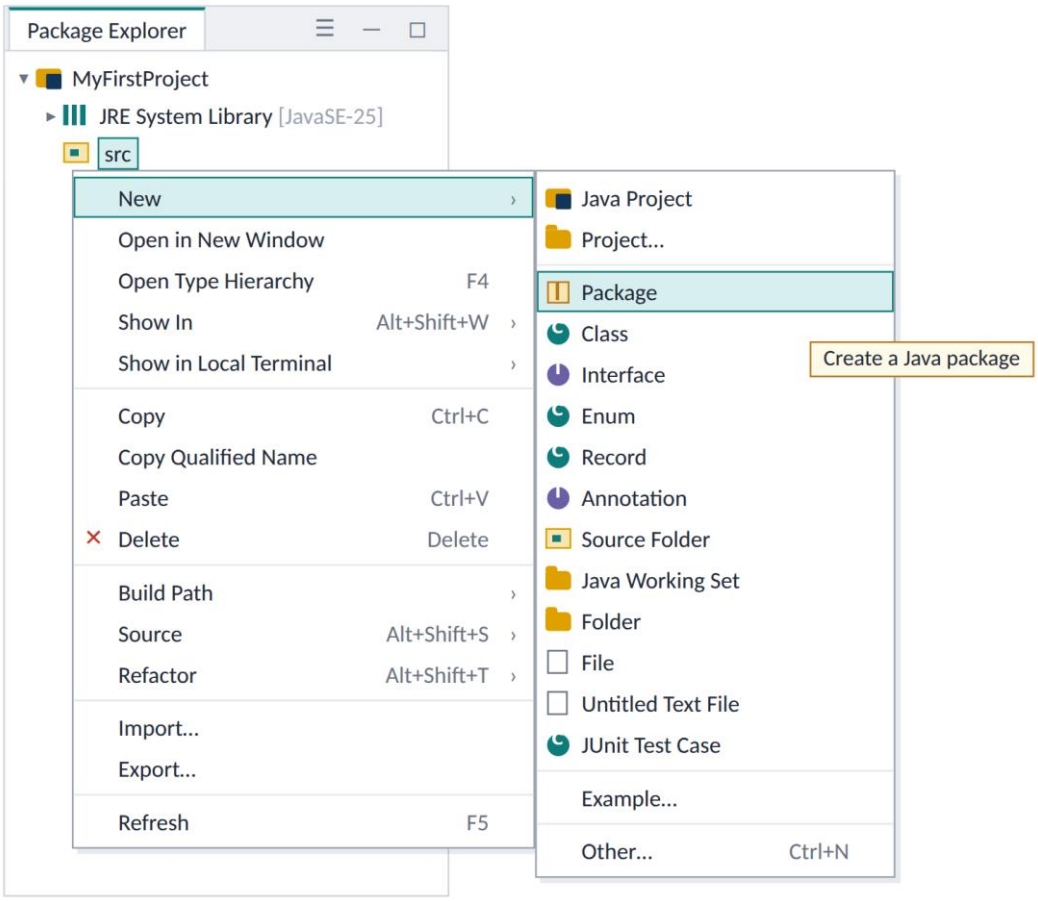


Figure 2.15 Right-click on src, then New > Package.

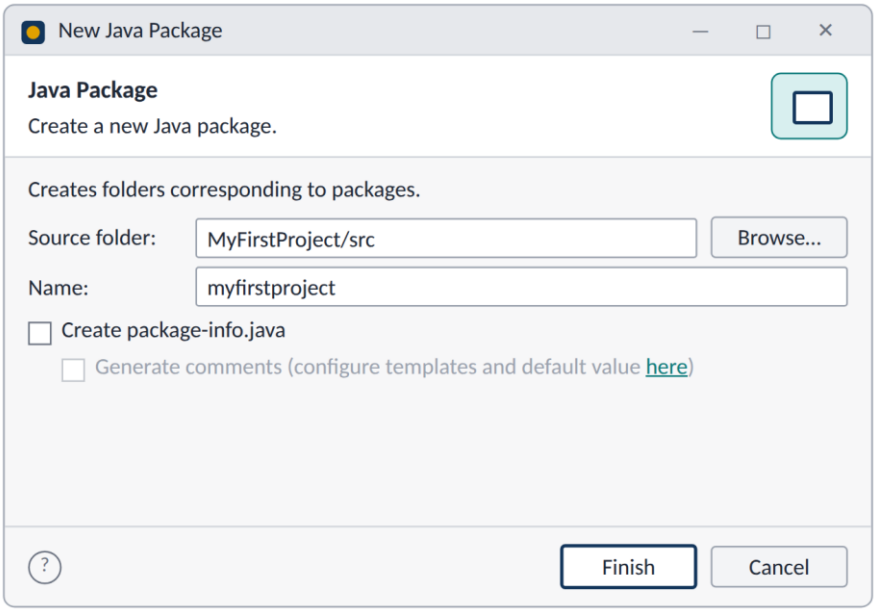


Figure 2.16 The New Java Package dialog: the source folder and the package name.

2.6.3 Creating a class and running it

Right-click the new package and choose *New > Class*. In the dialog of **Figure 2.17**, type `MyFirstClass` as the name, tick the stub **public static void main(String[] args)**, and press **Finish**. You could also type a new package name directly in this dialog; Eclipse would then create the package for you.

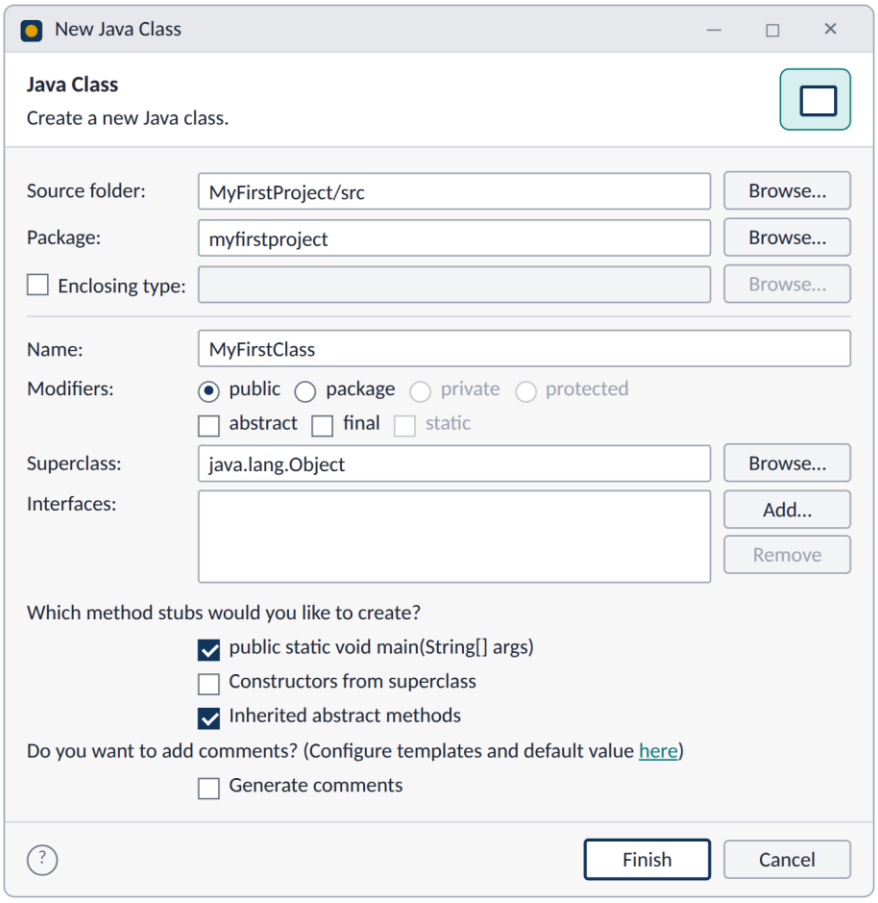


Figure 2.17 The New Java Class dialog, with the main method stub ticked.

Eclipse creates the file and opens it with a `// TODO Auto-generated method stub` comment inside `main`. Replace that comment with one line, so that the file reads as **Listing 2.3**.

Listing 2.3 `MyFirstClass.java` in package `myfirstproject`: the first program of the Eclipse project.

```
package myfirstproject;

public class MyFirstClass {

    public static void main(String[] args) {
        System.out.println("Hello Eclipse!");
    }
}
```

The first line, `package myfirstproject;`, says which package the class belongs to; it must match the folder the file is in (`src/myfirstproject`). The rest is the familiar shape from Chapter 1: a public class whose name matches the file name, and a `main` method where the program starts.

To run it, right-click the class in the Package Explorer, or right-click anywhere inside its editor, and choose *Run As > Java Application* (**Figure 2.18**). Eclipse saves the file if necessary, compiles it (it has actually been compiling in the background every time you saved), starts a JVM, and shows the program's output in the **Console** view (**Figure 2.19**):

```
Hello Eclipse!
```

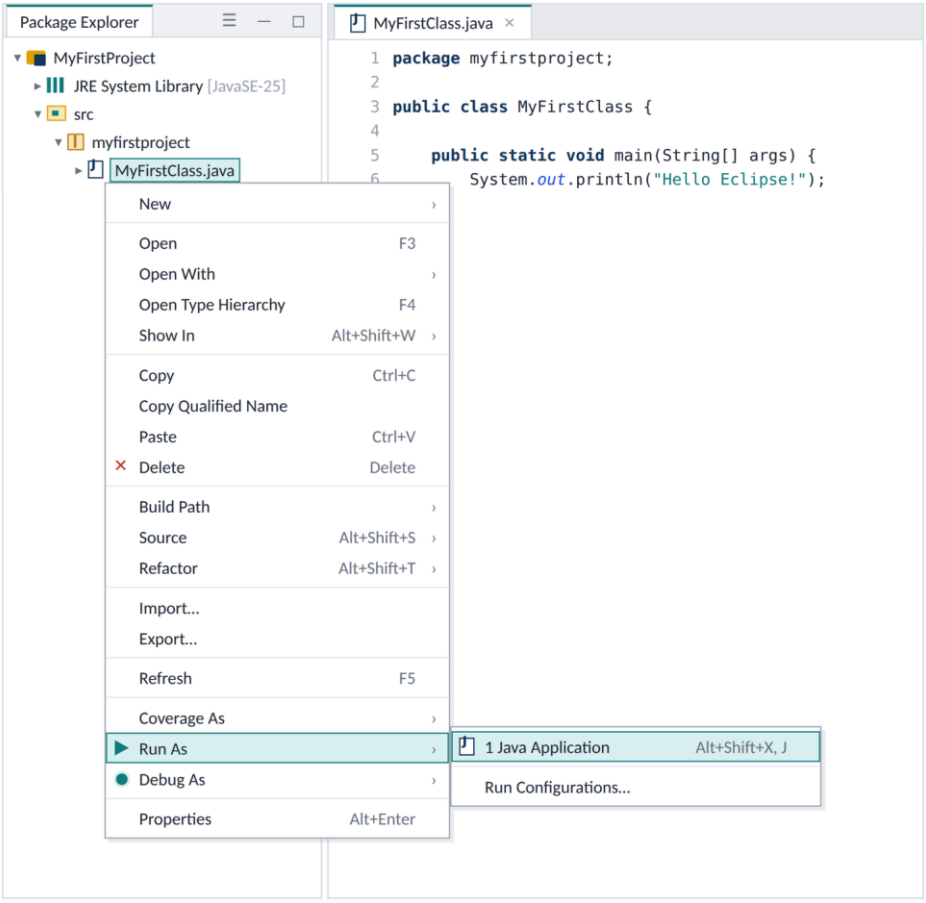


Figure 2.18 Run As > Java Application from the context menu of the class.

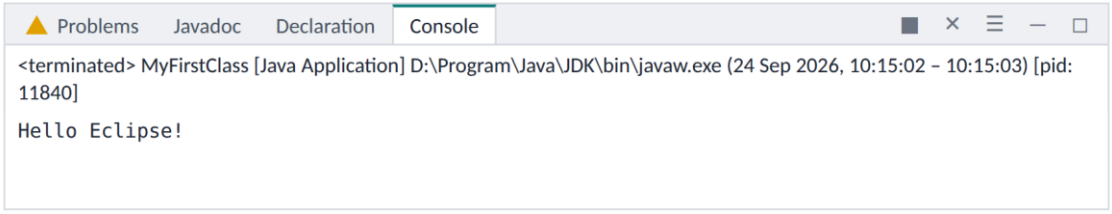


Figure 2.19 The Console view after the run. The first line records which program ran, on which Java, and when.

The first run creates a **launch configuration**, a saved record of which class to run, on which JRE, with which arguments; the green **Run** button repeats the most recent one, and *Run > Run Configurations...* edits them.

The Python twin is **Listing 2.4**. Python needs no project and no package for a single file: create a file called `my_first_module.py` in any folder, type the program, and run it.

Listing 2.4 `my_first_module.py`: the same first program in Python.

```
"""my_first_module.py: the first program of Chapter 2."""


def main():
    print("Hello VS Code!")


if __name__ == "__main__":
    main()
```

```
Hello VS Code!
```

The last two lines deserve a word, because you will see them in every Python program in this book. When Python runs a file directly, it sets the special variable `__name__` to the string `"__main__"`; when the same file is *imported* by another module, `__name__` is the module's own name instead. The test `if __name__ == "__main__":` therefore means "call `main()` only when this file is run as a program, not when it is imported." It is Python's equivalent of Java's rule that execution starts in `main`.

2.6.4 Running the program outside Eclipse: the JAR file

Your users will not install Eclipse to run your program. The standard way to distribute a Java program is a **JAR file** (Java ARchive): a single file, in ZIP format, that holds the compiled `.class` files and a small description file called the *manifest*.

To create one in Eclipse, right-click the project and choose *Export…*, then *Java* > *JAR file* and **Next** (**Figure 2.20**). In the next page (**Figure 2.21**) tick the project, choose a destination such as `myprogram1.jar` in the workspace folder, and press **Finish**.

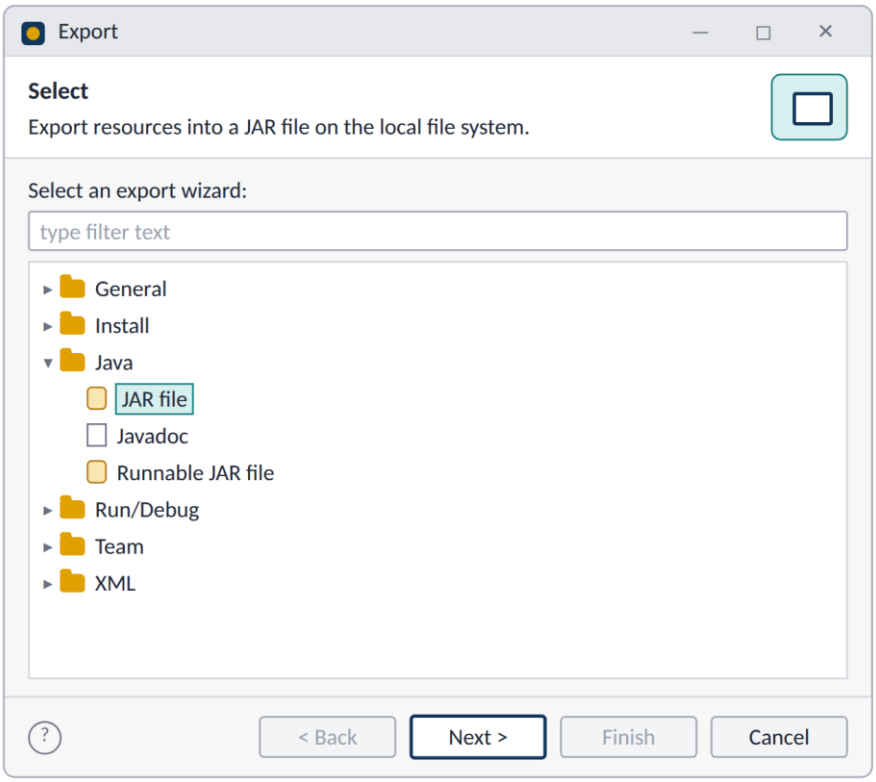


Figure 2.20 The Export wizard: JAR file under the Java category.

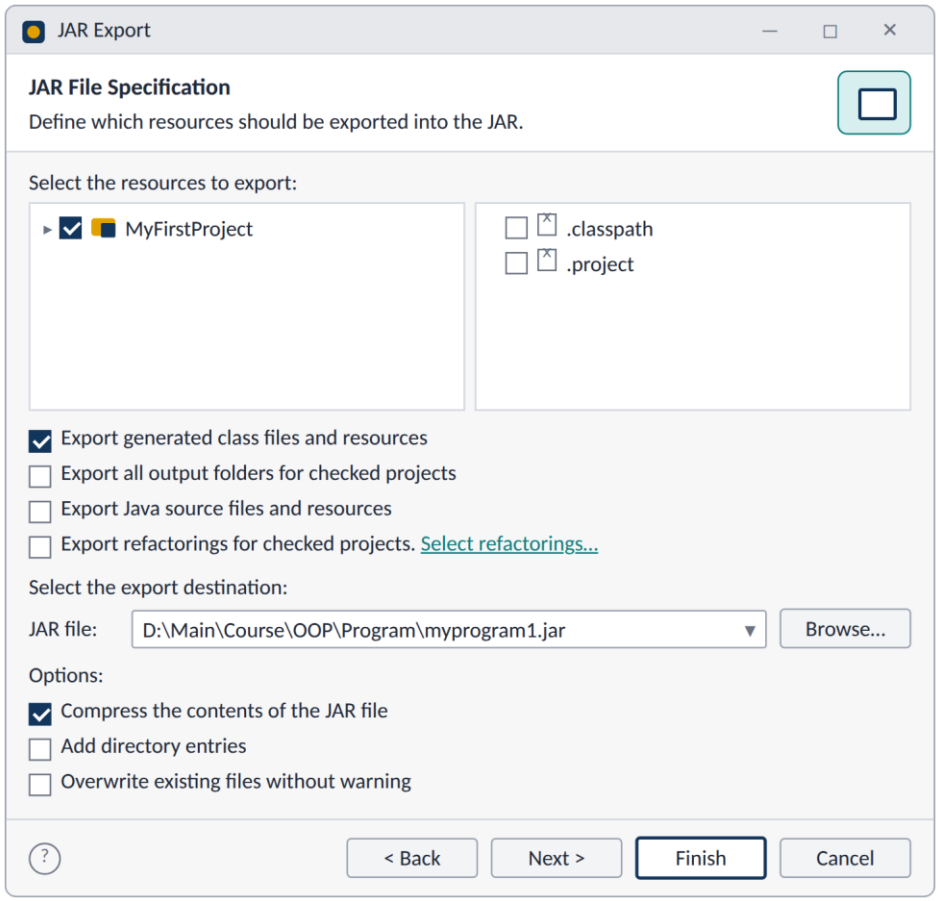


Figure 2.21 The JAR File Specification page: which resources to export and where to write the JAR.

Now open a terminal (on Windows, press the Windows key, type `cmd` and press Enter), change to the folder that holds the JAR with `cd`, and run the class from the JAR. The **classpath** is the list of places in which the JVM looks for classes; the option `-classpath` (or its short form `-cp`) puts the JAR on it, and the last word names the class to run, *including its package*:

```
java -classpath myprogram1.jar myfirstproject.MyFirstClass
```

```
Hello Eclipse!
```

Figure 2.22 shows the same command in a Windows Command Prompt. If you are curious about what is inside the JAR, the JDK's `jar` tool lists it with `jar tf myprogram1.jar` [6]:

```
META-INF/
META-INF/MANIFEST.MF
myfirstproject/
myfirstproject/MyFirstClass.class
```

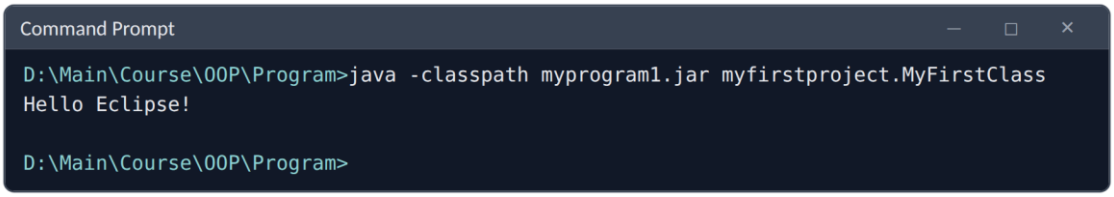


Figure 2.22 Running the class from the JAR in a Windows Command Prompt.

Notice the folder `myfirstproject/` inside the JAR: the package of the class has become a folder, exactly as in the `src` folder. That is not a coincidence, as Section 2.8 explains. (You can also make a

runnable JAR, whose manifest names the main class so that `java -jar myprogram1.jar` is enough; the Export wizard offers it as *Runnable JAR file*.)

2.7 The Same Work in VS Code

Everything you have just done in Eclipse has a direct counterpart in VS Code, and because VS Code treats Java and Python alike, it is the tool this book uses day to day. **Figure 2.23** is a map of the VS Code window during a debugging session; keep it in view while you read this section and Section 2.10.8.

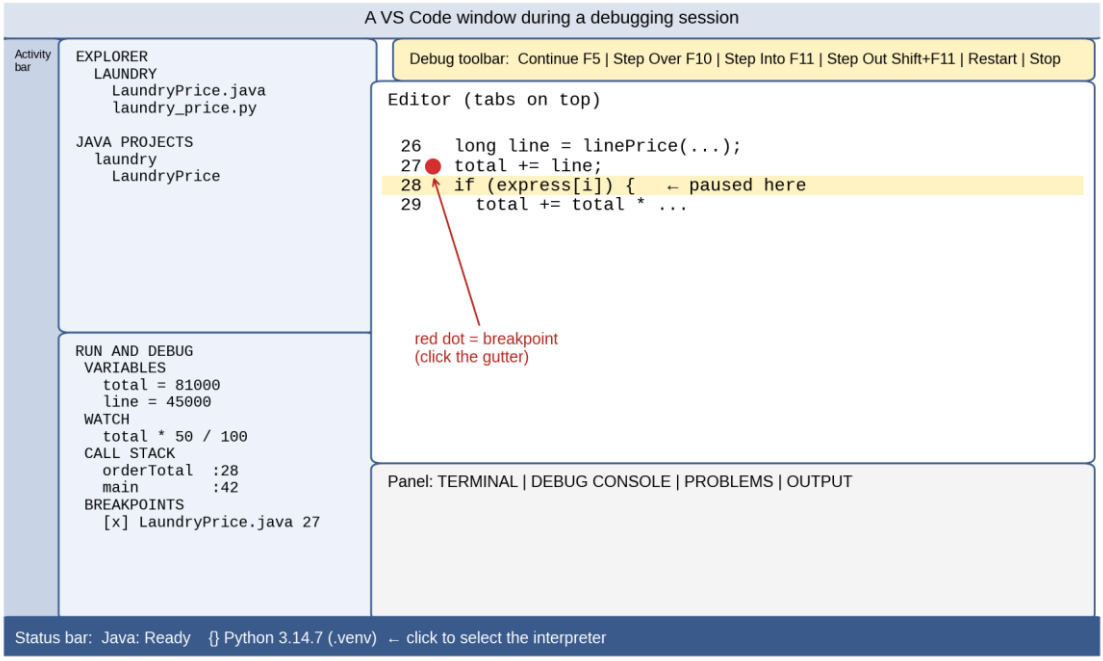


Figure 2.23 A map of the VS Code window while debugging: Explorer and Java Projects in the side bar, the editor with a breakpoint in the gutter, the Run and Debug views, the debug toolbar, the panel, and the interpreter selector in the status bar.

2.7.1 Folders instead of workspaces

VS Code has no start-up workspace dialog. Instead you open a **folder** with *File > Open Folder...*, and that folder is your working area: the **Explorer** view in the side bar shows its files, and settings specific to it are stored in a hidden subfolder called `.vscode`. If you need several folders at once, *File > Add Folder to Workspace...* creates a *multi-root workspace*, saved as a `.code-workspace` file; for this book one folder per chapter is enough.

A good arrangement is to open the folder `Code_new/Chapter02` that comes with this book, which contains a `java` folder and a `python` folder. When VS Code first opens a folder containing code it may ask whether you trust the folder; answer yes for your own folders, because an untrusted folder runs in a restricted mode in which the debugger is disabled.

2.7.2 Java in VS Code: the Java Projects view

The Extension Pack for Java recognizes a plain folder of `.java` files (an *unmanaged* project), an Eclipse project (it reads `.project` and `.classpath`), and Maven or Gradle projects [3]. The **Java Projects** view, at the bottom of the Explorer side bar, shows each project's packages, classes and

libraries, much like Eclipse's Package Explorer; its **+** button creates a project, and a right-click on a package offers *New Java Class*.

To run a Java program, open the file containing `main`. Two small links, **Run** and **Debug**, appear just above the `main` method; they are called *CodeLens* links. Clicking **Run** compiles the program and runs it in the **Terminal** panel at the bottom of the window. The status bar shows *Java: Ready* once the Java language server has finished reading your project; if it says *Java: Loading* for a long time, wait, because completion and error marks are not available until it is ready.

Which JDK compiles and runs your code? VS Code uses the JDK it finds through the `JAVA_HOME` environment variable or on the `PATH`, and the command *Java: Configure Java Runtime* (open the Command Palette with Ctrl+Shift+P, or Cmd+Shift+P on macOS, and start typing the command's name) shows and changes the choice for each project.

2.7.3 Python in VS Code: choosing the interpreter

A computer often has several Pythons: yours, the operating system's, and one inside each *virtual environment* (a private folder of Python plus packages for one project). Many "module not found, but I installed it!" problems come from running under a different Python from the one you think.

The rule is simple. Look at the right-hand end of the **status bar** (**Figure 2.23**): it shows the selected interpreter, for example *Python 3.14.7 (.venv)*. Click it, or run *Python: Select Interpreter* from the Command Palette, and choose the one you want from the list [4]. The choice is remembered for the folder.

To create a virtual environment, run *Python: Create Environment* and choose *Venv*; VS Code creates and selects a `.venv` folder (from a terminal: `python3 -m venv .venv`, or `py -m venv .venv` on Windows).

To run a Python file, open it and click the **Run Python File** triangle at the top right of the editor, or right-click in the editor and choose *Run Python File in Terminal*. VS Code types the full command into the Terminal panel for you, including the full path of the selected interpreter, so the Terminal also tells you exactly which Python ran your program.

2.7.4 Run configurations: `launch.json`

When a program needs special settings, such as command-line arguments, a particular working folder, or running a *module* with `python -m`, you describe them in a **launch configuration**, VS Code's name for the same idea as Eclipse's. Open the **Run and Debug** view (the triangle-with-a-bug icon in the activity bar, or Ctrl+Shift+D), click *create a launch.json file*, and choose Java or Python. VS Code writes a file `.vscode/launch.json`. Here is one with three configurations: one for the Java class `main.Tester` of Section 2.8, one for the current Python file, and one that runs the Python package module `main.tester` from the `python` folder:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "java",
            "name": "Tester (Java)",
            "request": "launch",
            "mainClass": "main.Tester"
        },
        {
            "type": "debugpy",
            "name": "Python: current file",
            "request": "launch",
            "program": "${file}",
            "console": "integratedTerminal"
        },
        {
            "type": "debugpy",
            "name": "Python: main.tester module",
            "request": "launch",
            "module": "main.tester",
            "cwd": "${workspaceFolder}/python"
        }
    ]
}
```

The `name` of each configuration appears in the drop-down list at the top of the Run and Debug view; choose one and press F5 to start it under the debugger, or Ctrl+F5 to run it without debugging. Expressions such as `${file}` (the file in the active editor) and `${workspaceFolder}` (the opened folder) are filled in by VS Code when the configuration starts. The type `debugpy` names the Python debugger that the Python extension uses; older configurations may say `python`, which still works.

2.7.5 The terminal is always there

Both IDEs run your programs with the same command-line tools as Section 2.3, and VS Code shows the exact command in its Terminal panel. Whenever the IDE surprises you, copy that command and run it yourself.

2.8 Projects, Packages, Modules, and Imports

2.8.1 Why group code at all

A program of fifty thousand lines, like the web application behind Profitina (Section 2.14), is spread over hundreds of files. Without grouping, *names collide* (two people both write a class called `Counter`) and *nobody can find anything*.

Both Java and Python solve these problems the same way: files are grouped into named **packages**, packages can contain other packages, and the full name of a thing includes the name of the package it lives in. A class called `Counter` in the package `util` has the full name `util.Counter`; a class called `Counter` in the package `org.w3c.dom.css` (there is one in the Java library) has the full name `org.w3c.dom.css.Counter`. The two no longer collide.

> **DEFINITION.** In Java, a *package* is a named group of classes, declared by a `package` statement on the first line of each file and stored in a folder of the same name. In Python, a *module* is a single `.py` file, and a *package* is a folder of modules, usually marked by a file called `__init__.py`. In both languages an *import* statement makes a name from another package usable in the current file.

2.8.2 Packages are folders

The rule that makes packages easy to understand is: **a package is a folder**. In Java, a class declared `package util;` must be in a file inside a folder called `util` (below the source folder `src`), and a class in the package `com.profitina.laundry` must be in `src/com/profitina/laundry/`. The compiled `.class` files follow the same folder structure in the output folder (`bin` in Eclipse), and in a JAR, as you saw at the end of Section 2.6.4.

In Python, a package is also a folder: the folder `util` containing `counter.py` is the package `util`, and the module inside it is `util.counter`. The folder usually contains a file called `__init__.py`, which may be empty. Its presence says "this folder is a regular package", and any code in it runs when the package is first imported. Since Python 3.3 a folder without `__init__.py` still works as a *namespace package*, but beginners should always add the file: it makes the intention explicit and avoids some confusing corner cases [8].

Naming conventions differ slightly. Java package names are all lowercase, often built from a reversed domain name for code that will be shared (`com.profitina.laundry`), and class names start with a capital letter (`Counter`). Python module and package names are short, all lowercase, and may use underscores (`laundry_price`); class names start with a capital letter as in Java [9], [8].

2.8.3 A project with two packages

We now build a slightly larger project, in which one class uses another from a different package. The project is called `counter` and has two packages: `util`, holding a class `Counter` that adds up numbers, and `main`, holding a class `Tester` that checks `Counter` works.

In Eclipse: create a Java project called `counter`; right-click `src` and create the package `util`, then the package `main`; right-click `util` and create the class `Counter`, and right-click `main` and create the class `Tester` (this time *with* the `main` method stub). In VS Code: in the `java` folder, create the folders `util` and `main` and the two files inside them; the Java Projects view shows them as packages automatically.

The Java project and its Python twin then look like this on the disk:

```
java/                              python/
  util/                              util/
    Counter.java                       __init__.py      (empty)
  main/                                counter.py
    Tester.java                      main/
  bin/            (compiled output)    __init__.py      (empty)
    util/Counter.class                 tester.py
    main/Tester.class
```

Listing 2.5 is the class `Counter`. Its method `count(x)` adds $1 + 2 + \cdots + x$ into a field called `result` and returns it; it refuses values of `x` outside the range 1 to 255 by *throwing an exception*, Java's way of saying "I cannot do this" (Chapter 5 treats exceptions fully).

Listing 2.5 `Counter.java` in package `util`: adds $1 + 2 + \cdots + x$ for $1 \le x \le 255$.

```java
package util;

public class Counter {

    private int result = 0;

    public int getResult() {
        return result;
    }

    // Adds 1 + 2 + ... + x; x must lie between 1 and 255.
    public int count(int x) {
        if (x > 0 && x <= 255) {
            for (int i = 1; i <= x; i++) {
                result += i;
            }
        } else {
            throw new RuntimeException("x should be between 1 and 255");
        }
        return result;
    }
}
```

In mathematical notation, `count(x)` computes

$$S(x) = \sum_{i=1}^{x} i = \frac{x(x+1)}{2}, \qquad 1 \le x \le 255, \tag{2.1}$$

so that $S(5) = 1 + 2 + 3 + 4 + 5 = 15$, which agrees with $\frac{5 \cdot 6}{2} = 15$. Equation (2.1) gives us an expected value to test against, which is exactly what `Tester` does. (When this class is first created from a template, the body of `count` contains only `TODO` comments and `return 0;`, as in **Figure 2.24**; completing it is the small exercise that the finished version shows. Eclipse collects every `TODO` comment in its *Tasks* view, which makes such comments a simple to-do list.)

Counter.java × Tester.java

```java
package util;

public class Counter {
    public int count(int x) {
        // TODO check that x > 0 and <= 255
        // if not throw a new RuntimeException
        // Example for a RuntimeException:

        // throw new RuntimeException("x should be between 1 and 255");

        // TODO calculate the numbers from 1 to x
        // for example if x is 5, calculate
        // 1 + 2 + 3 + 4 + 5


        // TODO return your calculated value
        // instead of 0
        return 0;
    }

}
```

Figure 2.24 The starting version of Counter.java: TODO comments describe what count(x) must do, and it returns 0 until they are done.

Listing 2.6 is the class `Tester`. It creates a `Counter`, checks that `count(5)` gives 15, and then checks that `count(256)` is refused, by catching the exception.

Listing 2.6 `Tester.java` in package `main`: a small hand-made test of `Counter`.

```java
package main;

import util.Counter;

public class Tester {

    public static void main(String[] args) {
        Counter counter = new Counter();
        int result = counter.count(5);
        if (result == 15) {
            System.out.println("Correct");
        } else {
            System.out.println("False");
        }
        try {
            counter.count(256);
        } catch (RuntimeException e) {
            System.out.println("Works as expected");
        }
    }
}
```

```
Correct
Works as expected
```

The line `import util.Counter;` is what lets `Tester`, which lives in package `main`, refer to `Counter` by its short name. Without it, `Tester` would have to write the full name `util.Counter` every time.

The Python twins are **Listing 2.7** and **Listing 2.8**. The class is written with Python's syntax (Chapter 3 explains `class`, `__init__` and `self` properly; for now read `self._result` as "this

object's `result`"), and the refusal is a `ValueError`, Python's standard exception for a value that is out of range.

Listing 2.7 `util/counter.py`: the Python twin of `Counter.java`.

```
"""util/counter.py: the Python twin of Counter.java."""


class Counter:

    def __init__(self):
        self._result = 0

    def get_result(self):
        return self._result

    # Adds 1 + 2 + ... + x; x must lie between 1 and 255.
    def count(self, x):
        if 0 < x <= 255:
            for i in range(1, x + 1):
                self._result += i
        else:
            raise ValueError("x should be between 1 and 255")
        return self._result
```

Listing 2.8 `main/tester.py`: the Python twin of `Tester.java`.

```
"""main/tester.py: checks util.counter.Counter; run it with python -m main.tester"""
from util.counter import Counter


def main():
    counter = Counter()
    result = counter.count(5)
    if result == 15:
        print("Correct")
    else:
        print("False")
    try:
        counter.count(256)
    except ValueError:
        print("Works as expected")


if __name__ == "__main__":
    main()
```

Python's import names the *module* and then the thing inside it: `from util.counter import Counter` means "from the module `counter` in the package `util`, bring in the name `Counter`". To run a module that is inside a package, go to the folder that *contains* the packages (here `python`) and use the `-m` switch with the module's dotted name, without `.py`:

```
cd python
python3 -m main.tester
```

```
Correct
Works as expected
```

Why not simply `python3 main/tester.py`? Because Python looks for imported packages starting from the folder of the script it was given. Started as `main/tester.py`, it would look for `util` inside `main`, not find it, and stop with `ModuleNotFoundError: No module named 'util'`. Started with `-m` from the `python` folder, it looks in the `python` folder, where `util` is. This single rule explains most "module not found" errors that beginners meet.

2.8.4 The missing import, in both languages

Now let us make the mistake that Section 2.3.5 promised: delete the import line from each tester and see how each language reacts.

In Eclipse, the moment the import is gone, `Counter` is underlined in red in the editor, a red marker appears in the left ruler, and the Problems view lists the error. Compiling from a terminal shows exactly what the IDE is reacting to:

```
javac -d bin src/util/Counter.java src/main/Tester.java
```

```
src/main/Tester.java:7: error: cannot find symbol
        Counter counter = new Counter();
        ^
  symbol:   class Counter
  location: class Tester
src/main/Tester.java:7: error: cannot find symbol
        Counter counter = new Counter();
                          ^
  symbol:   class Counter
  location: class Tester
2 errors
```

The compiler found the problem at *compile time*, before anything ran: the name `Counter` on line 8 of the original file (line 7 now, because the import line is gone) cannot be found in package `main`. It reports it twice, once for the type in `Counter counter` and once for the constructor in `new Counter()`, with a caret under each.

The IDE offers to repair this for you. Hover the mouse over the red underline (or click the red marker in the ruler) and a box lists *quick fixes* (**Figure 2.25**). The first one, *Import 'Counter' (util)*, adds exactly the line we deleted. Notice the second suggestion, *Import 'Counter' (org.w3c.dom.css)*: the Java library really does contain another class called `Counter`, and only you know which one you meant. This is the name collision of Section 2.8.1, and the reason why packages exist.

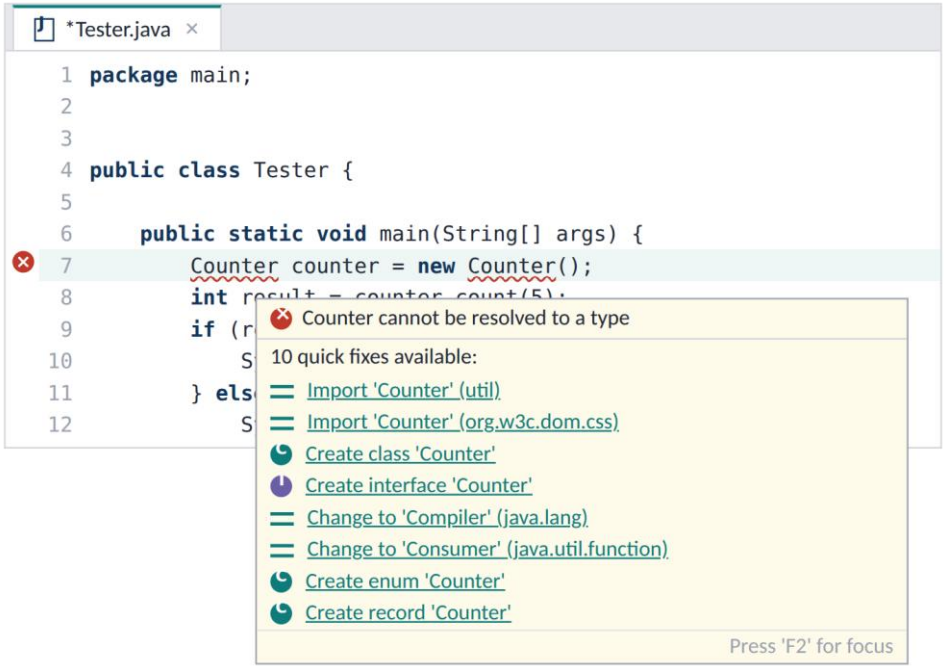


Figure 2.25 Counter cannot be resolved to a type: the quick-fix box offers to import util.Counter, or the unrelated org.w3c.dom.css.Counter.

Eclipse has a second way to add imports, which is better when several are missing at once: *Source > Organize Imports* (Ctrl+Shift+O, or Cmd+Shift+O on macOS). It adds every import that can be worked out without asking, removes imports that are not used, and sorts the rest. When a short name matches several classes, it asks, as in **Figure 2.26**. VS Code offers the same command as *Organize Imports* (Shift+Alt+O) and the same quick fixes under the light-bulb icon (Ctrl+.).

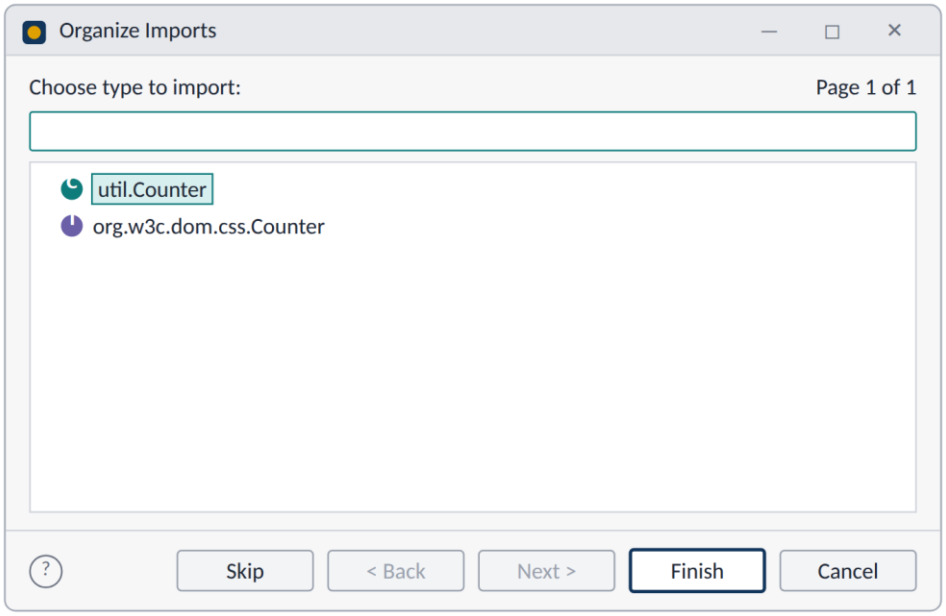


Figure 2.26 Organize Imports asks which Counter is meant when the short name is ambiguous.

Python, without the import, starts running and fails only when it reaches the line that uses the name:

```
Traceback (most recent call last):
  File "<frozen runpy>", line 198, in _run_module_as_main
  File "<frozen runpy>", line 88, in _run_code
  File "/home/student/counter/py/main/tester.py", line 18, in <module>
    main()
  File "/home/student/counter/py/main/tester.py", line 5, in main
    counter = Counter()
              ^^^^^^^
NameError: name 'Counter' is not defined. Did you mean: 'counter'?
```

This is a *run-time* error: the program started, called `main()`, and only on line 5 discovered that `Counter` does not exist. (The `<frozen runpy>` frames belong to the machinery behind `python -m`.) The suggestion `counter` is a guess, not an answer. VS Code's Python extension underlines the undefined name before you run, but that is the editor helping you, not the language.

2.8.5 Compiling a project with packages by hand

When a Java project has packages, the terminal commands of Section 2.3.3 need two small additions. From the folder that contains `src`:

```
javac -d bin src/util/Counter.java src/main/Tester.java
java -cp bin main.Tester
```

The option `-d bin` tells `javac` to write the `.class` files below the folder `bin`, creating one subfolder per package. Listing `bin` afterwards shows the result:

```
bin
bin/main
bin/main/Tester.class
bin/util
bin/util/Counter.class
```

The option `-cp bin` puts `bin` on the classpath, and the class to run is named with its package, `main.Tester`, never with a slash or the `.class` extension. The output is the same as in Eclipse:

```
Correct
Works as expected
```

These two lines are exactly what Eclipse and VS Code do for you when you press *Run*.

2.8.6 Exporting, importing, and navigating projects

Exporting and importing. To give a whole Eclipse project to someone else, or to move it to another workspace, use *File > Export > General > Archive File*, tick the projects, and choose a `.zip` file (**Figure 2.27**). To bring projects in, use *File > Import > General > Existing Projects into Workspace*, choose *Select archive file* (or *Select root directory* if the zip has already been unpacked), tick the projects, and press **Finish** (**Figure 2.28**). A project can be imported only if no project with the same name already exists in the workspace; if you are re-importing your own project, make sure the zip exists before you delete the old copy. VS Code needs none of this: a project is just a folder, so zipping and unzipping the folder is enough.

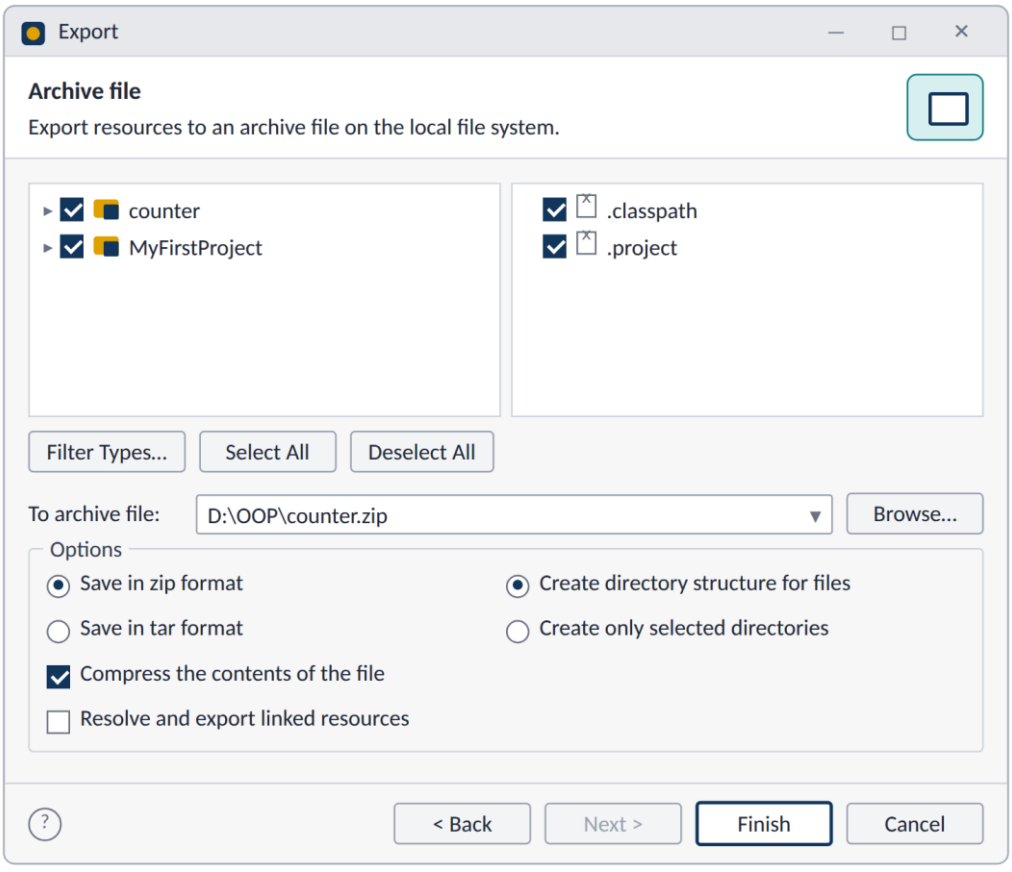


Figure 2.27 Exporting two projects to an archive file, keeping the directory structure.

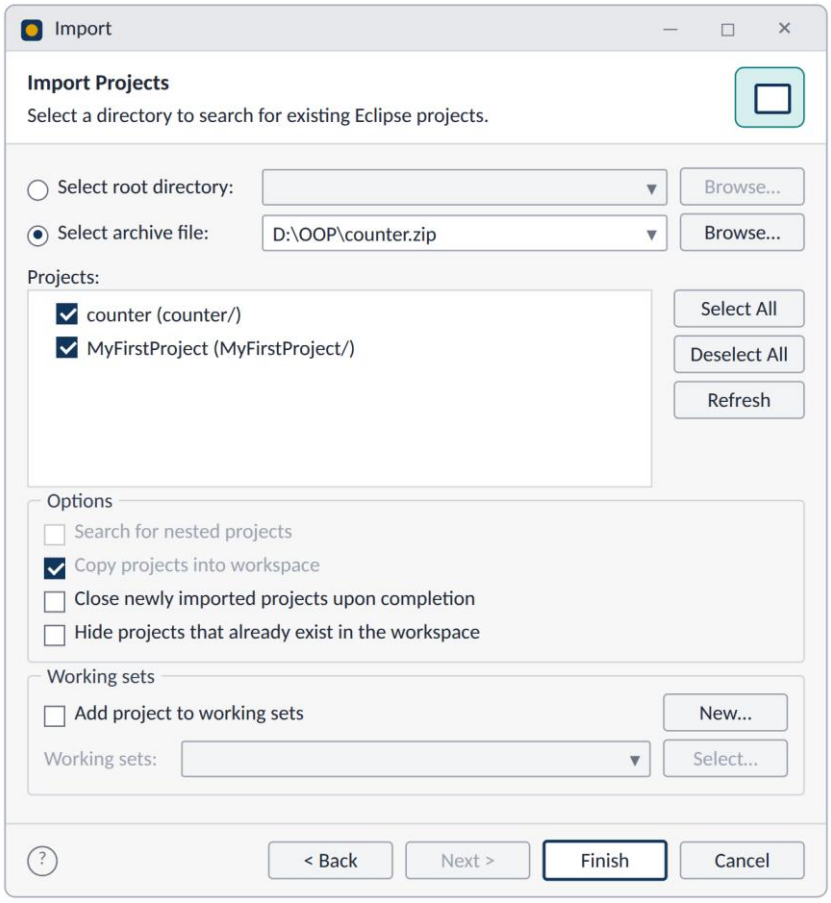


Figure 2.28 Importing existing projects from an archive file into the workspace.

Linking the explorer with the editor. In a large project it is easy to lose track of *where* the file you are editing lives. The Package Explorer's view menu (the three dots at its top right) has an item **Link with Editor** (**Figure 2.29**); when it is on, switching to the editor of `Counter.java` selects `Counter.java` in the tree. Its *Filters...* item hides kinds of files you do not want to see. VS Code does the same automatically: the Explorer follows the active editor unless you turn off the setting *Explorer: Auto Reveal*.

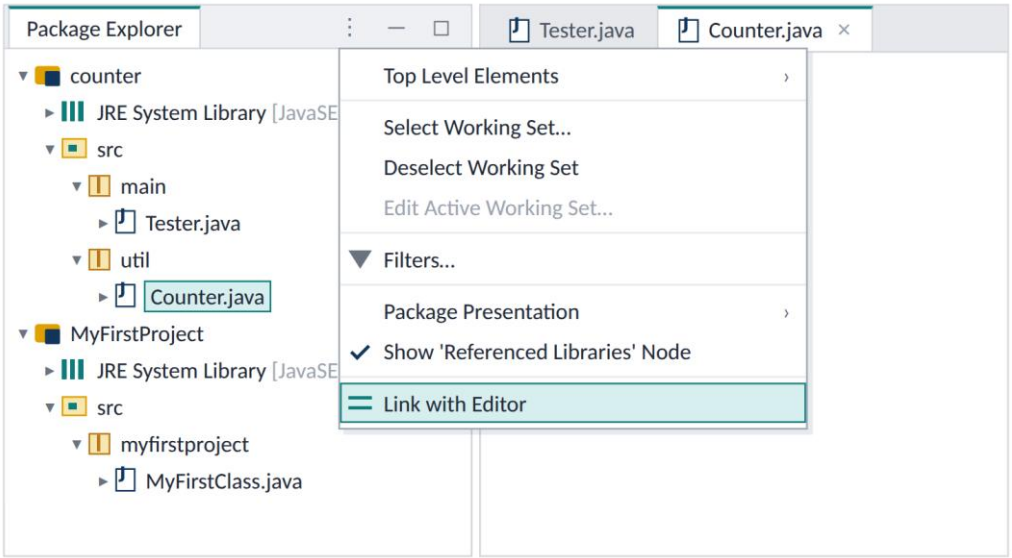


Figure 2.29 The Package Explorer's view menu, with Link with Editor about to be switched on.

2.9 Editing Power Tools: Content Assist, Quick Fix, Refactoring, and Formatting

The editor of a good IDE does far more than let you type. Four of its tools will save you hours in the very first week.

Content assist (IntelliSense). Type the start of a name and press Ctrl+Space: the editor lists every variable, method, class and keyword that fits at that point, with its type and documentation, and inserts the one you choose. After a dot it lists the members of the object before the dot, so `counter.` followed by Ctrl+Space lists `count(int)` and `getResult()`. Content assist also expands *templates*: in Eclipse, type `sysout` or `main` and press Ctrl+Space to get `System.out.println();` or a complete `main` method; in VS Code, type `sysout` or `main` and pick the suggestion. In Python, typing `if __name__` and choosing the suggestion produces the whole `if __name__ == "__main__":` line.

Quick Fix. Whenever something is underlined, press Ctrl+1 in Eclipse or Ctrl+. in VS Code (Cmd+1 and Cmd+. on macOS). You saw it add a missing import; it can also create a method you called but have not written yet, add a missing `return`, change a variable's type, or surround a risky call with the `try` and `catch` that Java demands.

Refactoring. To *refactor* is to change the structure of code without changing what it does [10]. The most important refactoring is **Rename**: put the cursor on `count`, choose *Refactor > Rename* in Eclipse (Alt+Shift+R) or press F2 in VS Code, type `sumUpTo`, and the IDE changes the declaration and every use in every file, and nothing else, because it knows which `count` is which. The same menu offers *Extract Method*, *Extract Local Variable*, *Inline* and *Move* (to another package, updating all imports); VS Code offers the same for Python.

Formatting. *Source > Format* in Eclipse (Ctrl+Shift+F) and *Format Document* in VS Code (Shift+Alt+F) re-indent and re-space a whole file according to a consistent style. Format before you ask anyone, including yourself tomorrow, to read your code: badly indented code hides bugs.

Table 2.3 collects the keys worth learning now. The Eclipse keys are for Windows and Linux; on macOS, Ctrl usually becomes Cmd.

Table 2.3 Keyboard shortcuts worth learning in the first week, in Eclipse and in VS Code.

Action	Eclipse	VS Code
Content assist / completion	Ctrl+Space	Ctrl+Space
Quick fix	Ctrl+1	Ctrl+.
Organize imports	Ctrl+Shift+O	Shift+Alt+O
Rename everywhere	Alt+Shift+R	F2
Format the file	Ctrl+Shift+F	Shift+Alt+F
Go to the definition	F3	F12
Find a file or class by name	Ctrl+Shift+R / Ctrl+Shift+T	Ctrl+P
Toggle comment on lines	Ctrl+/	Ctrl+/
Run the last program	Ctrl+F11	Ctrl+F5
Debug the last program	F11	F5
Toggle a breakpoint on the line	Ctrl+Shift+B	F9

WARNING. Notice the trap in the last rows of **Table 2.3**: in Eclipse, F11 *starts* debugging and F5 *steps into*; in VS Code, F5 *starts* (or continues) debugging and F11 steps into. If you use both tools, say the action to yourself ("step into") rather than trusting your fingers.

2.10 Debugging from Zero

2.10.1 What a bug is, and two ways to look for one

A **bug** is a difference between what a program does and what it should do. Loud bugs stop the compiler or the program with a message that says where to look (Section 2.13). Silent bugs are harder: the program runs to the end and prints a reasonable-looking number that is wrong. A laundry bill of Rp 109,350 looks just as believable as one of Rp 93,150.

There are two classic ways of looking inside a program to find a silent bug.

The first is **print debugging**: add `System.out.println(...)` or `print(...)` lines that show variables at interesting moments, run, and read the output. It works anywhere, but every new question means editing and running again, you must guess in advance *what* to print and *where*, and the extra lines must be removed afterwards.

The second is the **debugger**, which runs your program under its control so that you can pause it at any line, see *every* variable, move one statement at a time into or out of method calls, and even change values, all without editing the program.

DEFINITION. *Debugging* is the process of finding and removing the cause of a bug. A *debugger* is a tool that runs a program interactively, so that you can stop it at chosen points, execute it step by step, and inspect and modify its state while you watch the source code.

2.10.2 The vocabulary

Seven words are enough to talk about any debugger.

- A **breakpoint** is a mark on a line that says "stop *before* running this line"; both IDEs show it as a dot in the editor's left ruler.
- A program stopped at a breakpoint is **suspended**; the line about to run is highlighted, and nothing runs until you say so.
- **Stepping** executes a suspended program a little at a time: *step over*, *step into* or *step out* (Section 2.10.4).
- **Resume** (*continue*) runs freely to the next breakpoint or the end.
- The **variables** of the current method are shown with their values while the program is suspended.
- A **watch** is an expression, such as `total * 50 / 100`, re-evaluated at every stop.
- The **call stack** lists the methods currently running, each called by the one below it.

A relative of the breakpoint, the **watchpoint**, stops whenever a particular *field* is read or modified, whichever line does it; Eclipse sets one by double-clicking in the ruler beside a field declaration. Breakpoints and watchpoints together are *stop points*.

2.10.3 Breakpoints and starting the debugger in Eclipse

We use the `counter` project of Section 2.8. Open `Tester.java`. To set a breakpoint, right-click in the left ruler next to a line and choose **Toggle Breakpoint** (**Figure 2.30**), or simply double-click in the ruler at that position, or put the cursor on the line and press Ctrl+Shift+B. Set one on the

line `Counter counter = new Counter();`. A small blue dot appears in the ruler (**Figure 2.31**). Doing the same again removes it; that is what *toggle* means.

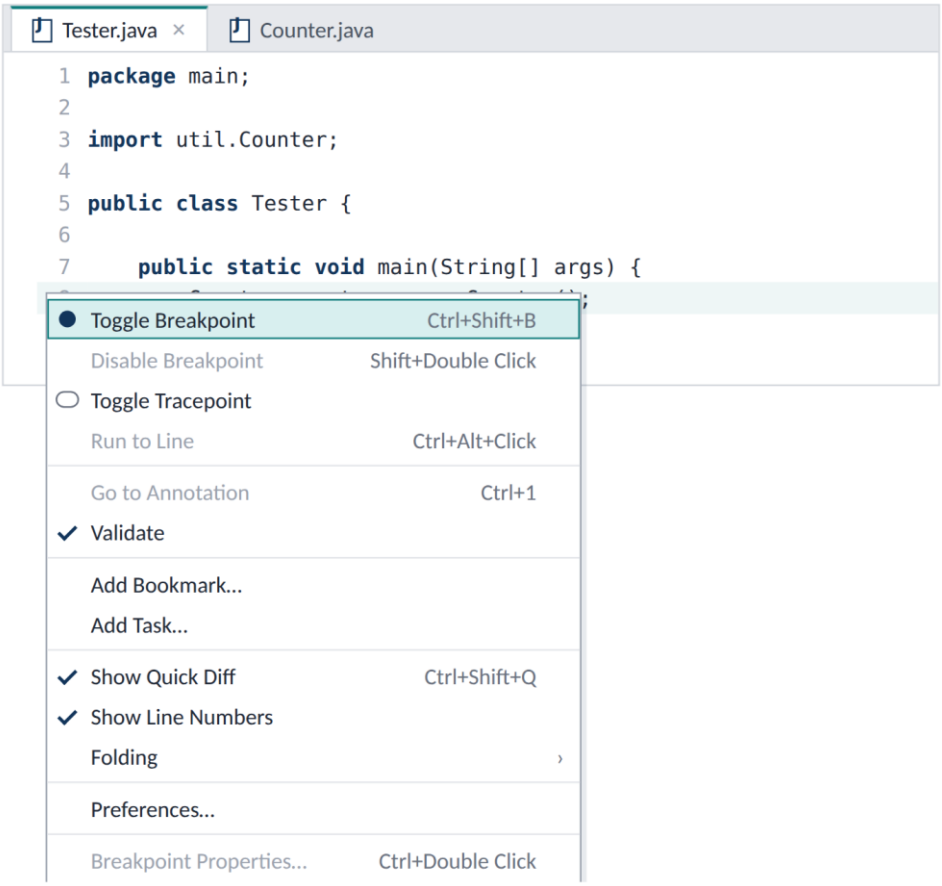


Figure 2.30 Toggle Breakpoint in the context menu of the editor's left ruler (Ctrl+Shift+B).

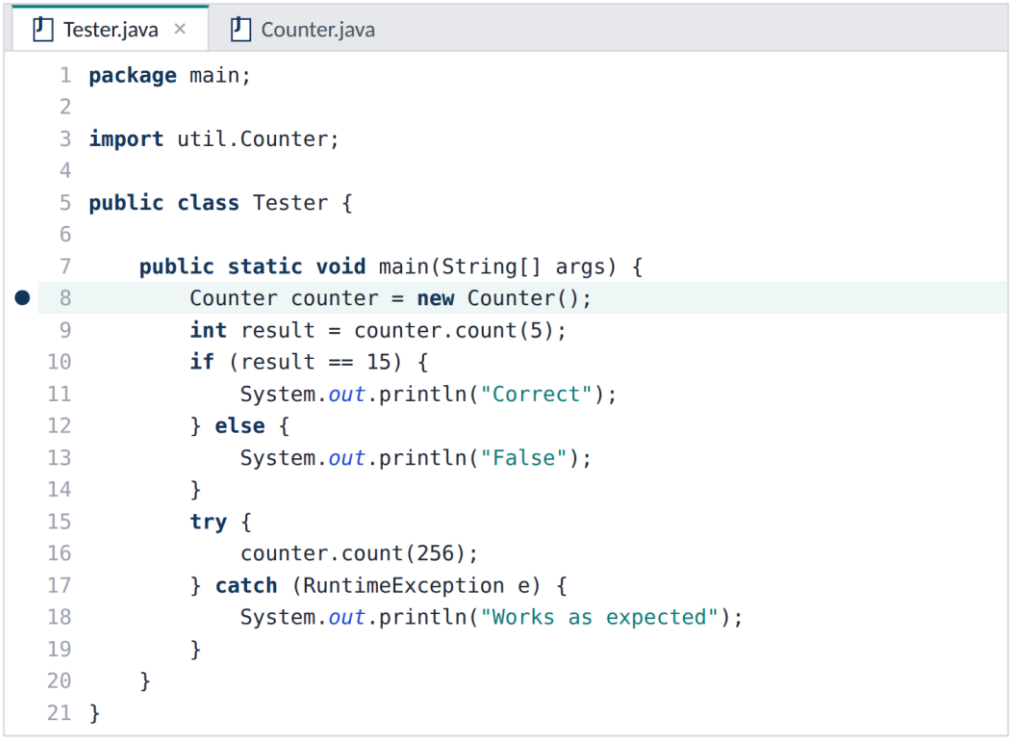


Figure 2.31 A breakpoint on line 8, Counter counter = new Counter(); the dot in the ruler marks it.

Running the program normally ignores breakpoints. To run it *under the debugger*, right-click the class (or inside its editor) and choose **Debug As > Java Application** (**Figure 2.32**). Once you have done this, the bug-shaped **Debug** button on the toolbar starts the same launch configuration again (**Figure 2.33**).

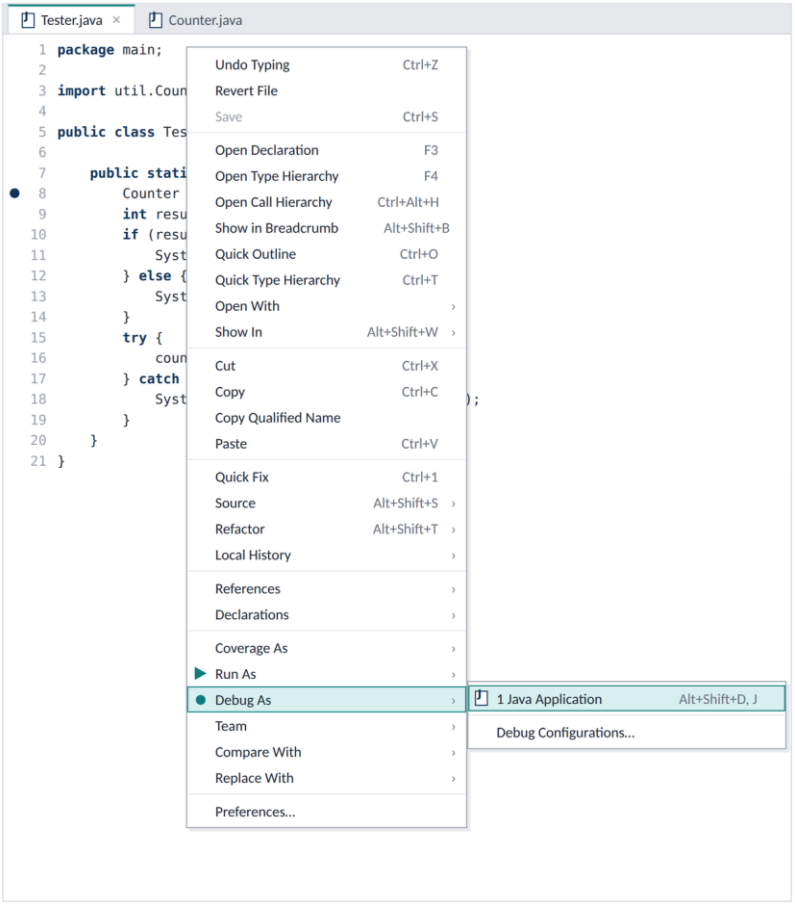


Figure 2.32 Debug As > Java Application starts the program under the debugger.

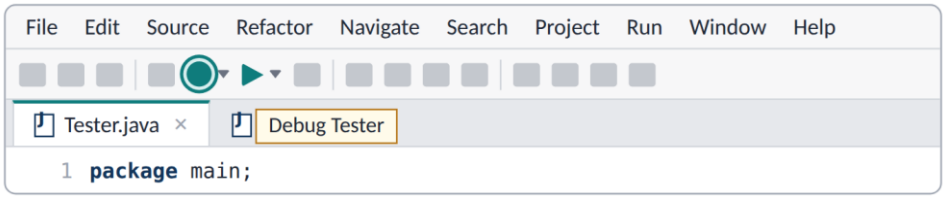


Figure 2.33 The Debug toolbar button reruns the most recent debug launch, here Tester.

When execution reaches the breakpoint, Eclipse asks whether it should switch to the Debug perspective (**Figure 2.34**). Answer **Switch** (and tick *Remember my decision* so that it stops asking). If you have set no breakpoints, the program simply runs to the end, exactly as with *Run As*: the debugger stops only where you have asked it to.

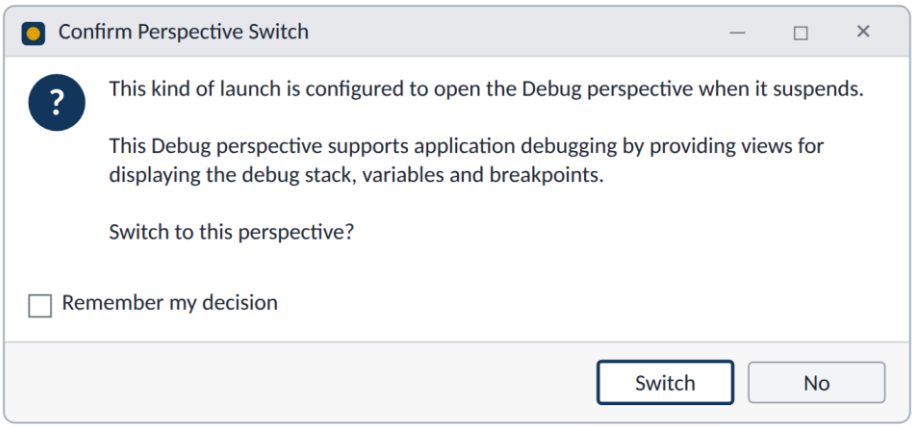


Figure 2.34 Eclipse offers to open the Debug perspective when the program first suspends.

2.10.4 Controlling execution: resume, step over, step into, step out

The Debug perspective's toolbar has buttons for controlling the suspended program, and each has a function key (**Figure 2.35**). It is much faster to learn the keys.

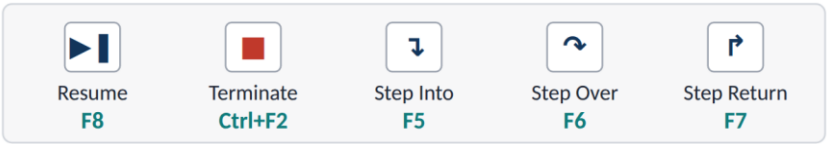


Figure 2.35 The execution buttons of the Eclipse debugger with their keys: Resume (F8), Terminate (Ctrl+F2), Step Into (F5), Step Over (F6) and Step Return (F7).

Table 2.4 gives the meaning of each control and its key in both IDEs and in the two terminal debuggers used in Section 2.11, `jdb` for Java and `pdb` for Python.

Table 2.4 The debugger's execution controls and their keys and commands in Eclipse, VS Code, jdb and pdb.

Control	What it does	Eclipse	VS Code	Jdb	Pdb
Step Into	Execute the current line; if it calls a method of yours, stop at the first line inside that method	F5	F11	`step`	`step`
Step Over	Execute the current line completely, including any method it calls, and stop at the next line of the same method	F6	F10	`next`	`next`
Step Out (Step Return)	Finish the current method and stop in the caller, just after the call	F7	Shift+F11	`step up`	`return`
Resume (Continue)	Run until the next breakpoint, or to the end	F8	F5	`cont`	`continue`
Terminate (Stop)	End the program at once	Ctrl+F2	Shift+F5	`exit`	`quit`

The three steps are easiest to understand as movements on the call stack, as in **Figure 2.36**. Suppose the program is suspended in a method `orderTotal` on a line that calls `linePrice(...)`, which in turn calls `pricePerKg(...)`. **Step Over** treats the call as a single action: `linePrice` and everything it calls run to completion, and the program stops on the next line of `orderTotal`. **Step Into** goes *down* into the call and stops on the first line of `linePrice`. **Step Out** does the opposite: it lets the current method finish and stops back in the method that called it.

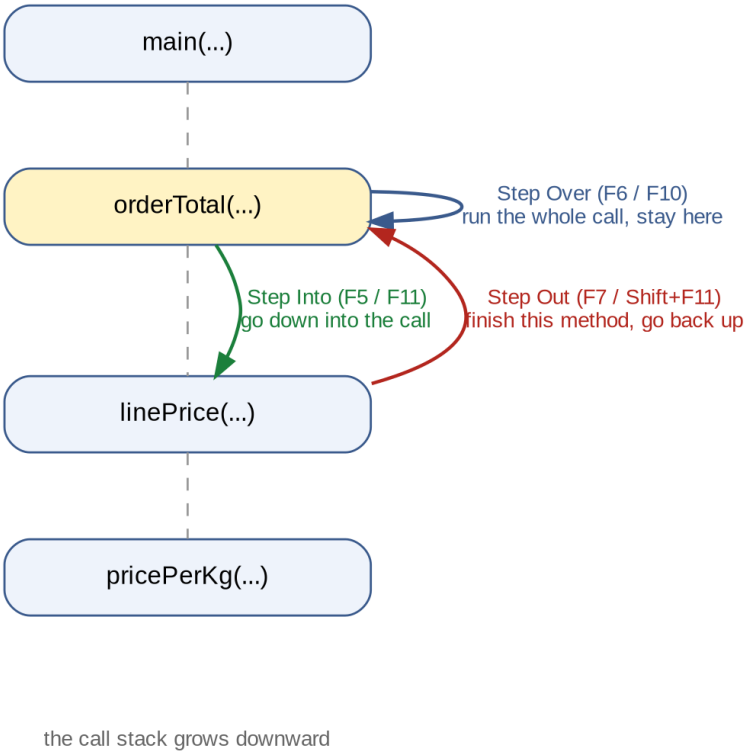


Figure 2.36 Stepping seen on the call stack. Step Into goes down into the called method, Step Out returns to the caller, and Step Over runs the whole call and stays at the same level.

A rule of thumb: *step over* code you trust, *step into* code you suspect, and *step out* as soon as you realize you have stepped into something uninteresting. Library methods such as `Math.max` or `System.out.println` are normally skipped by Step Into, because both IDEs apply *step filters* that keep you out of the Java library and, in VS Code's Python debugger, out of code that is not your own (the `justMyCode` setting). That is almost always what you want.

2.10.5 The call stack

> **DEFINITION.** The *call stack* is the list of method calls that have started but not yet finished. The method currently executing is on top; below it is the method that called it, then the one that called that, down to `main` (and, below `main`, the runtime's own start-up code).

Eclipse shows the call stack in the **Debug** view, at the top left of the Debug perspective (**Figure 2.37**). The tree reads: the launch (`Tester [Java Application]`), the JVM it started, the thread `main` (suspended), and under the thread one line per stack *frame*, here only `Tester.main(String[]) line: 11`. Clicking a lower frame shows, in the editor and the Variables view, the line and the variables of *that* method at the moment it made its call; this is how you find out *why* a method was called with the arguments it received.

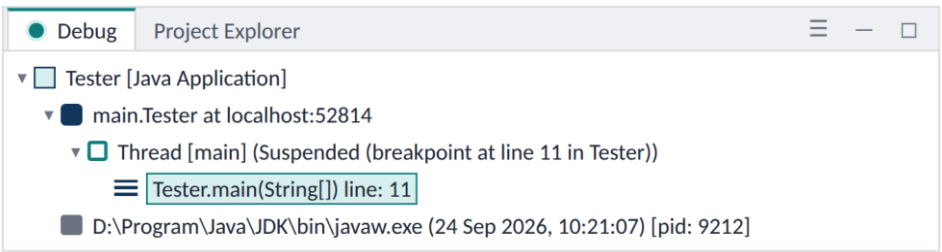


Figure 2.37 The Debug view shows the thread main suspended in Tester.main at line 11; each line under the thread is one frame of the call stack.

It is called a *stack* because it behaves like a stack of plates: each call puts a frame on top, each return takes the top frame off, and only the top one is active.

2.10.6 Looking at variables, and changing them

The **Variables** view, at the top right of the Debug perspective, lists the parameters and local variables of the selected frame while the program is suspended (**Figure 2.38**). A value changed by the last step is highlighted, the fastest way to see what a step did, and expanding an object such as `counter` shows its fields. The view's menu can also show static variables and, under *Layout > Select Columns...*, each value's *actual type*, useful from Chapter 7 on.

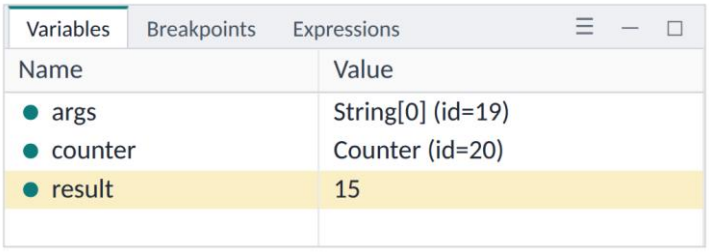


Figure 2.38 The Variables view after count(5) returned: args, counter and result = 15.

You can also *change* a value while the program is suspended: click the value in the Variables view, type a new one, and press Enter (**Figure 2.39**). In VS Code, right-click a variable and choose *Set Value*; in `pdb` simply type an assignment such as `!total = 0`. This lets you ask "what if?" questions without editing the program: what if the order were not express, what if the customer were not a member? The change lasts only for this run.

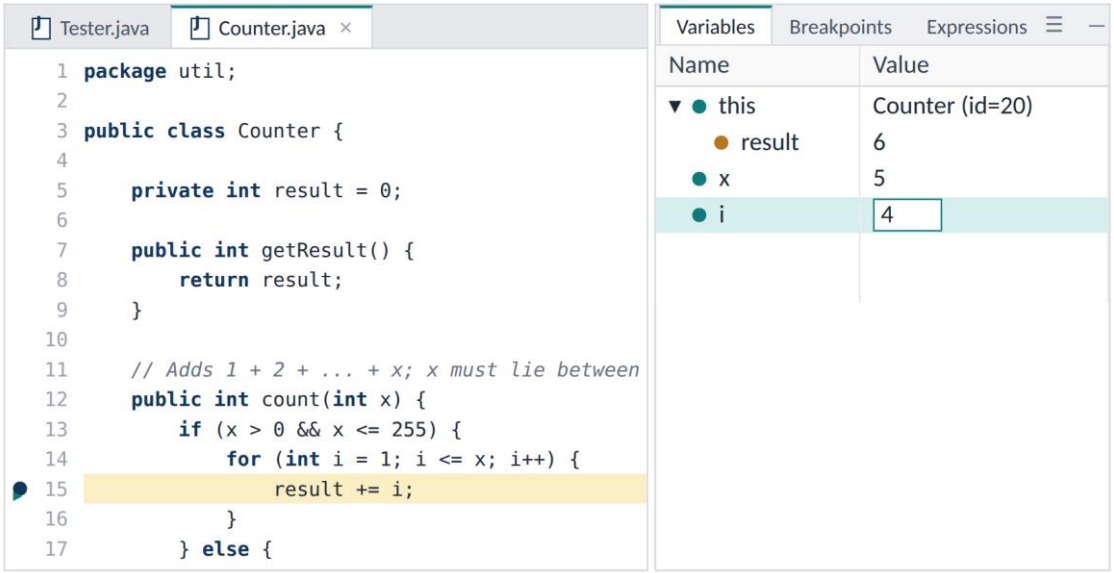


Figure 2.39 Changing a value during debugging: the loop variable i in count(int) is being edited in the Variables view.

How an object is displayed. By default the Variables view shows an object using its `toString()` method. For a class that does not define one, this gives something like `util.Counter@71bbf57e`, the class name and a number, which tells you nothing (**Figure 2.40**). Chapter 3 shows how to write `toString()`; until then, Eclipse lets you define a **Detail Formatter**: right-click the variable, choose *New Detail Formatter...*, and type a Java expression to display instead, such as `getResult()` (**Figure 2.41**). From then on every `Counter` is displayed as its result.

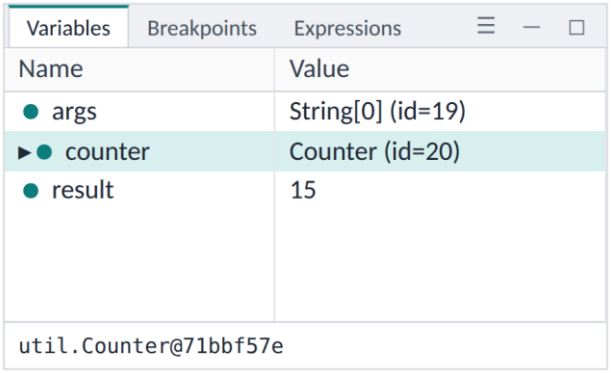


Figure 2.40 Without a toString() method, the detail pane shows only util.Counter@71bbf57e.

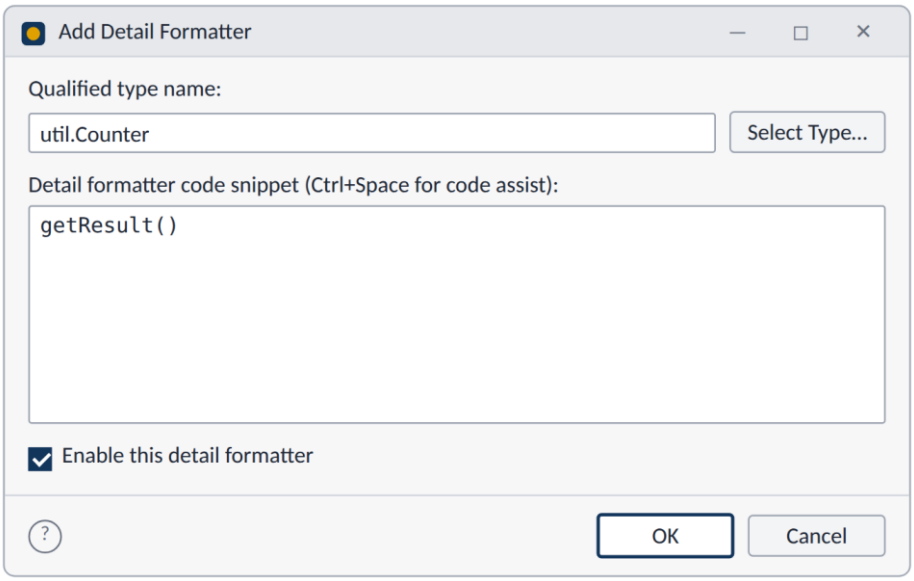


Figure 2.41 A Detail Formatter for util.Counter that displays getResult() instead.

Watches and expressions. The *Expressions* view (Eclipse) and the *Watch* section (VS Code) hold expressions that are re-evaluated at every stop. Right-click a variable or a selected expression in the editor and choose *Watch*, or click the **+** in the Watch section and type, for example, `total * 50 / 100`. Section 2.11 uses a watch to see a wrong value before it is added to the total.

2.10.7 Managing breakpoints: disabling, skipping, hit counts, and conditions

After an hour of debugging you may have a dozen breakpoints in several files. The **Breakpoints** view (**Figure 2.42**) lists them all. Unticking a breakpoint's box *disables* it without deleting it, so you can switch it back on later; the **X** buttons delete the selected breakpoint or all of them. The bottom of the view shows the properties of the selected breakpoint, and two of them are powerful:

- **Hit count**: stop only on the *n*-th time the line is reached. In a loop that runs 1,000 times, a hit count of 500 takes you straight to the middle.
- **Conditional**: stop only when a Boolean expression that you type is true, for example `i == 7` or `billedKg < kg`. Section 2.12 is devoted to this.

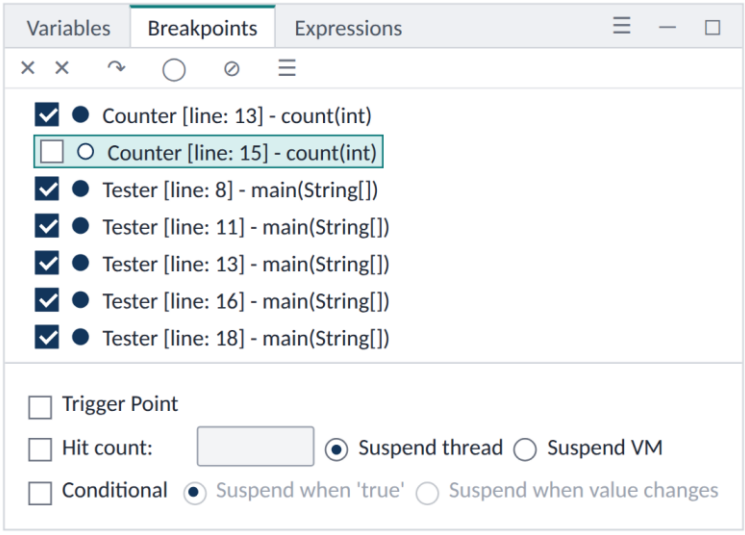


Figure 2.42 The Breakpoints view lists every breakpoint; the check boxes enable or disable them, and the lower part holds the Hit count and Conditional properties.

The **Skip All Breakpoints** button (Ctrl+Alt+B) in the same view's toolbar (**Figure 2.43**) turns every breakpoint off at once; pressing it again turns them all back on. It is useful when you want the program to run freely for a while without losing your carefully placed breakpoints.

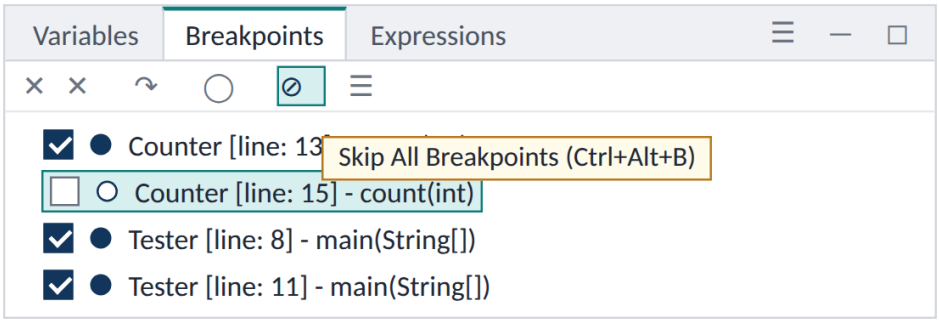


Figure 2.43 Skip All Breakpoints (Ctrl+Alt+B) disables every breakpoint without deleting any.

2.10.8 The same debugger in VS Code, for Java and for Python

VS Code's debugger works the same way for both languages, and **Figure 2.23** shows its parts [3], [4].

- **Setting a breakpoint.** Click in the gutter to the left of a line number; a red dot appears. F9 toggles a breakpoint on the cursor's line.
- **Starting.** For Java, click the **Debug** CodeLens above `main`, or press F5. For Python, choose *Python Debugger: Debug Python File* from the drop-down next to the Run triangle at the top right of the editor, or press F5 and pick *Python File*. If a `launch.json` exists, F5 starts its selected configuration.
- **The views.** The **Run and Debug** side bar shows the sections *Variables*, *Watch*, *Call Stack* and *Breakpoints*, which correspond one to one to Eclipse's Variables, Expressions, Debug and Breakpoints views. The Breakpoints section also has check boxes for stopping on *Uncaught Exceptions* and *Raised Exceptions* (Python) or *Uncaught Exceptions* and *Caught Exceptions* (Java).
- **The toolbar.** A floating toolbar appears at the top of the editor with Continue (F5), Step Over (F10), Step Into (F11), Step Out (Shift+F11), Restart and Stop.
- **The Debug Console** evaluates any expression you type, such as `line * 50 // 100`, in the current frame.

- **Conditional breakpoints.** Right-click a breakpoint dot (or the gutter) and choose *Edit Breakpoint…* or *Add Conditional Breakpoint…*; choose *Expression*, *Hit Count* or *Log Message*. A *logpoint* (log message) prints a message such as `total is {total}` each time the line is reached, without stopping: a print statement that never needs to be deleted from the source.

> **KEY INSIGHT.** Every graphical debugger, in every IDE, for every language, offers the same small set of controls: breakpoints (plain, conditional, hit-count), resume, step over, step into, step out, variables, watches and the call stack. Learn the ideas once, on one tool, and you can use any debugger within minutes.

2.11 A Worked Hunt: The Laundry Bill That Was Too High

2.11.1 The symptom

Profitina Laundry, the small laundry business that runs through this book, charges by weight. Its price list is simple:

- *wash*: Rp 6,000 per kg; *iron*: Rp 4,000 per kg; *wash+iron*: Rp 9,000 per kg;
- a load lighter than 3 kg is charged as 3 kg (the machine runs anyway);
- *express* service costs 50% more on that item;
- a member whose bill reaches Rp 100,000 gets 10% off the whole bill.

Written as formulas, the price of item i, with service rate $r(s_i)$, weight w_i kilograms, and $e_i = 1$ if the item is express and $e_i = 0$ if it is not, is

$$p_i = \left(1 + \frac{e_i}{2}\right) \cdot r(s_i) \cdot \max(w_i, 3), \tag{2.2}$$

and the bill for n items is

$$T = \begin{cases} \frac{9}{10} P & \text{if the customer is a member and } P \geq 100\,000, \\ P & \text{otherwise,} \end{cases} \qquad \text{where } P \sum_{i=1}^{n} p_i. \tag{2.3}$$

Bu Sari, a member, brings three items: 4 kg to wash, 2 kg to iron, and 5 kg to wash and iron, the last one express. The cashier's program, **Listing 2.9** in Java and **Listing 2.10** in Python, computes her bill. It uses only what Chapter 1 and this chapter have introduced: constants, a few methods (functions in Python), arrays (lists in Python) holding one entry per item, and a loop. The program prints the total it computed and, for comparison, the total that Bu Sari's receipt from last week says it should be.

Listing 2.9 `LaundryPrice.java`: the cashier's price calculator, with a bug.

```java
// LaundryPrice.java: prices one Profitina Laundry order. This version has a bug.
public class LaundryPrice {

    static final double MIN_KG = 3.0;            // lighter loads are billed as 3 kg
    static final long EXPRESS_PERCENT = 50;      // express costs 50% more
    static final long MEMBER_DISCOUNT_PERCENT = 10;
    static final long DISCOUNT_THRESHOLD = 100_000;

    static long pricePerKg(String service) {
        return switch (service) {
            case "wash" -> 6_000;
            case "iron" -> 4_000;
            case "wash+iron" -> 9_000;
            default -> throw new IllegalArgumentException("unknown service: " + service);
        };
    }

    static long linePrice(String service, double kg) {
        double billedKg = Math.max(kg, MIN_KG);
        return (long) (billedKg * pricePerKg(service));
    }

    static long orderTotal(String[] services, double[] kgs, boolean[] express, boolean
member) {
        long total = 0;
        for (int i = 0; i < services.length; i++) {
            long line = linePrice(services[i], kgs[i]);
            total += line;
            if (express[i]) {
                total += total * EXPRESS_PERCENT / 100;
            }
        }
        if (member && total >= DISCOUNT_THRESHOLD) {
            total -= total * MEMBER_DISCOUNT_PERCENT / 100;
        }
        return total;
    }
```

Listing 2.9 (Continued) *LaundryPrice.java: the cashier's price calculator, with a bug.*

```java
    public static void main(String[] args) {
        String[] services = {"wash", "iron", "wash+iron"};
        double[] kgs = {4.0, 2.0, 5.0};
        boolean[] express = {false, false, true};
        long total = orderTotal(services, kgs, express, true);
        System.out.println("Customer : Bu Sari (member)");
        System.out.println("Total    : Rp " + total);
        System.out.println("Expected : Rp 93150");
    }
}
```

```
Customer : Bu Sari (member)
Total    : Rp 109350
Expected : Rp 93150
```

Listing 2.10 `laundry_price.py`: the same calculator in Python, with the same bug.

```
"""laundry_price.py: prices one Profitina Laundry order. This version has a bug."""

MIN_KG = 3.0                      # lighter loads are billed as 3 kg
EXPRESS_PERCENT = 50              # express costs 50% more
MEMBER_DISCOUNT_PERCENT = 10
DISCOUNT_THRESHOLD = 100_000


def price_per_kg(service):
    if service == "wash":
        return 6_000
    elif service == "iron":
        return 4_000
    elif service == "wash+iron":
        return 9_000
    raise ValueError("unknown service: " + service)


def line_price(service, kg):
    billed_kg = max(kg, MIN_KG)
    return int(billed_kg * price_per_kg(service))


def order_total(services, kgs, express, member):
    total = 0
    for i in range(len(services)):
        line = line_price(services[i], kgs[i])
        total += line
        if express[i]:
            total += total * EXPRESS_PERCENT // 100
    if member and total >= DISCOUNT_THRESHOLD:
        total -= total * MEMBER_DISCOUNT_PERCENT // 100
    return total


def main():
    services = ["wash", "iron", "wash+iron"]
    kgs = [4.0, 2.0, 5.0]
    express = [False, False, True]
    total = order_total(services, kgs, express, True)
    print("Customer : Bu Sari (member)")
    print("Total    : Rp", total)
    print("Expected : Rp 93150")
```

Listing 2.10 (Continued) *laundry_price.py: the same calculator in Python, with the same bug.*

```
if __name__ == "__main__":
    main()
```

```
Customer : Bu Sari (member)
Total    : Rp 109350
Expected : Rp 93150
```

Both programs agree with each other, and both are wrong: Bu Sari is being charged Rp 109,350 instead of Rp 93,150, an overcharge of Rp 16,200. This is the worst kind of bug, silent and plausible, and it is costing a real customer real money.

A few words about the code before we hunt. `static final` in Java (and, by convention, a name in capital letters in Python) marks a *constant*, a value that never changes while the program runs. The `switch` in `pricePerKg` chooses a return value by comparing `service` with each `case`; its `default` branch throws an exception for an unknown service, which Section 2.13 will provoke on purpose. The underscores in `100_000` are only for readability; the value is one hundred thousand.

`Math.max(kg, MIN_KG)` (Python: `max(kg, MIN_KG)`) is the larger of the two, which implements the 3 kg minimum. The cast `(long)` (Python: `int(...)`) turns the `double` product into a whole number of rupiah; for weights given in half kilograms, as here, the product is always a whole number, so nothing is lost.

2.11.2 Step 1: know what the answer should be

You cannot recognize a wrong value unless you know the right one. Before touching the debugger, work the bill out by hand with Equation (2.2) and Equation (2.3), one line per item, as in **Table 2.5**.

Table 2.5 Bu Sari's bill worked out by hand. These are the values the program should reach, item by item.

Item	Service	Weight w_i	Billed kg $\max(w_i,3)$	Rate $r(s_i)$	Base price	Express?	p_i
1	wash	4.0	4.0	6,000	24,000	no	24,000
2	iron	2.0	3.0	4,000	12,000	no	12,000
3	wash+iron	5.0	5.0	9,000	45,000	yes, +22,500	67,500
						P	103,500
						discount 10%	−10,350
						T	93,150

The table gives us more than the answer at the bottom: it gives the correct *intermediate* values. After item 1 the running total should be 24,000; after item 2, 36,000; after item 3, $36{,}000 + 67{,}500 = 103{,}500$. Any place where the program's running total departs from this column is where the bug is.

2.11.3 Step 2: form a hypothesis, then place a breakpoint

The wrong total is too *high*, by Rp 16,200. What could make a bill too high? The discount might be missing (but a missing discount would give 103,500, not 109,350); the minimum weight might be applied wrongly; the express surcharge might be too big. We do not know yet, and that is fine: a hypothesis is only a guess about *where to look*. The loop in `orderTotal` is where all the item prices come together, so that is where we put a breakpoint: on the line that calls `linePrice` (line 26 in Java, line 27 in Python).

In Eclipse, double-click in the ruler of line 26 and choose *Debug As > Java Application*; in VS Code, click the gutter of line 26 (Java) or 27 (Python) and press F5. The program stops before the call, with `i = 0` and `total = 0` in the Variables view.

2.11.4 Step 3: a guided tour with step into, step over and step out

First we check that `linePrice` prices item 1 correctly, and practice all three steps on the way: Step Into three times (into `linePrice`, over the library call `Math.max`, into `pricePerKg`), Step Out twice (back to `linePrice`, back to `orderTotal`), and Step Over once to complete the assignment. In the IDE you press Eclipse F5, F5, F5, F7, F7, F6 or VS Code F11, F11, F11, Shift+F11, Shift+F11, F10, and watch the Variables view and the call stack after each press.

A screenshot can show only one moment of this, but the JDK and Python both include plain *terminal* debuggers, `jdb` for Java [6] and `pdb` for Python [11], that print every step as text. Here is `jdb` (compile with `javac -g LaundryPrice.java` so that the class file keeps the names of local variables, then start `jdb -classpath . LaundryPrice`); the commands after the prompts `>` and `main[1]` are what was typed:

```
Initializing jdb ...
> stop at LaundryPrice:26
Deferring breakpoint LaundryPrice:26.
It will be set after the class is loaded.
> run
run LaundryPrice
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint LaundryPrice:26

Breakpoint hit: "thread=main", LaundryPrice.orderTotal(), line=26 bci=13
26                long line = linePrice(services[i], kgs[i]);

main[1] print i
 i = 0
main[1] step
>
Step completed: "thread=main", LaundryPrice.linePrice(), line=19 bci=0
19            double billedKg = Math.max(kg, MIN_KG);

main[1] locals
Method arguments:
service = "wash"
kg = 4.0
Local variables:
main[1] step
>
Step completed: "thread=main", LaundryPrice.linePrice(), line=20 bci=8
20            return (long) (billedKg * pricePerKg(service));

main[1] step
>
Step completed: "thread=main", LaundryPrice.pricePerKg(), line=10 bci=0
10            return switch (service) {

main[1] where
  [1] LaundryPrice.pricePerKg (LaundryPrice.java:10)
  [2] LaundryPrice.linePrice (LaundryPrice.java:20)
  [3] LaundryPrice.orderTotal (LaundryPrice.java:26)
  [4] LaundryPrice.main (LaundryPrice.java:42)
main[1] step up
>
Step completed: "thread=main", LaundryPrice.linePrice(), line=20 bci=13
20            return (long) (billedKg * pricePerKg(service));

main[1] step up
>
Step completed: "thread=main", LaundryPrice.orderTotal(), line=26 bci=24
26                long line = linePrice(services[i], kgs[i]);

main[1] next
>
Step completed: "thread=main", LaundryPrice.orderTotal(), line=27 bci=26
27                total += line;

main[1] print line
 line = 24000
main[1] clear LaundryPrice:26
Removed: breakpoint LaundryPrice:26
main[1] cont
> Customer : Bu Sari (member)
Total    : Rp 109350
Expected : Rp 93150

The application exited
```

Follow it line by line. `stop at LaundryPrice:26` sets the breakpoint; `jdb` says it is *deferred* because the class is not loaded yet. `run` starts the program, which stops at line 26 with `i = 0`. The first `step` enters `linePrice` at line 19, and `locals` shows its two parameters and no local variables yet. The next `step` runs line 19 (without entering `Math.max`), the one after enters `pricePerKg` at line 10, and `where` prints the call stack, top frame first: exactly the four frames of **Figure 2.36**, with the line each has reached. Two `step up` commands climb back out, one frame at a time, and `next` completes line 26, after which `print line` shows 24000. Finally `clear` removes the breakpoint and `cont` lets the program run to the end. (The stray `>` prompts on otherwise empty lines are `jdb` printing its prompt before the event it is waiting for arrives; they are harmless.)

The same tour in `pdb`, started with `python3 -m pdb laundry_price.py`, where the commands are typed after `(Pdb)`:

```
> /home/student/laundry/laundry_price.py(1)<module>()
-> """laundry_price.py: prices one Profitina Laundry order. This version has a bug."""
(Pdb) break 27
Breakpoint 1 at /home/student/laundry/laundry_price.py:27
(Pdb) continue
> /home/student/laundry/laundry_price.py(27)order_total()
-> line = line_price(services[i], kgs[i])
(Pdb) p i, total
(0, 0)
(Pdb) step
--Call--
> /home/student/laundry/laundry_price.py(19)line_price()
-> def line_price(service, kg):
(Pdb) step
> /home/student/laundry/laundry_price.py(20)line_price()
-> billed_kg = max(kg, MIN_KG)
(Pdb) p service, kg
('wash', 4.0)
(Pdb) step
> /home/student/laundry/laundry_price.py(21)line_price()
-> return int(billed_kg * price_per_kg(service))
(Pdb) p billed_kg
4.0
(Pdb) step
--Call--
> /home/student/laundry/laundry_price.py(9)price_per_kg()
-> def price_per_kg(service):
(Pdb) where
  /usr/lib/python3.11/bdb.py(600)run()
-> exec(cmd, globals, locals)
  <string>(1)<module>()
  /home/student/laundry/laundry_price.py(47)<module>()
-> main()
  /home/student/laundry/laundry_price.py(40)main()
-> total = order_total(services, kgs, express, True)
  /home/student/laundry/laundry_price.py(27)order_total()
-> line = line_price(services[i], kgs[i])
  /home/student/laundry/laundry_price.py(21)line_price()
-> return int(billed_kg * price_per_kg(service))
> /home/student/laundry/laundry_price.py(9)price_per_kg()
-> def price_per_kg(service):
(Pdb) return
--Return--
> /home/student/laundry/laundry_price.py(11)price_per_kg()->6000
-> return 6_000
(Pdb) return
--Return--
> /home/student/laundry/laundry_price.py(21)line_price()->24000
-> return int(billed_kg * price_per_kg(service))
(Pdb) next
> /home/student/laundry/laundry_price.py(28)order_total()
-> total += line
(Pdb) p line
24000
(Pdb) quit
```

`pdb` stops before the first line of the file, so we set the breakpoint on line 27 and `continue` to reach it. `step` enters `line_price` (pdb first shows `--Call--` on the `def` line, then the first statement). `p service, kg` prints values, and `p billed_kg` after the next step shows 4.0. The next `step` enters `price_per_kg`, and `where` shows the stack, this time *oldest frame first*, with the current frame marked `>` at the bottom. (The top two frames, in `bdb.py` and `<string>`, belong to `pdb` itself.) `pdb`'s step-out command is `return`: it runs until the current function is about to return and shows the value, `->6000` and then `->24000`. `next` completes the assignment and `p line` confirms 24000.

Item 1 is priced correctly. The bug is not in `linePrice` for this item, and we have learned where everything is. Now we hunt.

2.11.5 Step 4: watch the running total

Move the breakpoint to the line `total += line;` (Java line 27, Python line 28), so that the program stops once per item, just before the item is added. Add two watches: `total`, and the expression `total * EXPRESS_PERCENT / 100` (in Python, `total * EXPRESS_PERCENT // 100`), which is the surcharge the program is about to add when an item is express. Now press **Resume** (Eclipse F8, VS Code F5) to go from item to item, and compare each value with **Table 2.5**.

- Stop 1: `line = 24000`, `total = 0`. Correct.
- Stop 2: `line = 12000`, `total = 24000`. Correct: the 2 kg of ironing was billed as 3 kg.
- Stop 3: `line = 45000`, `total = 36000`, and `express[i]` is `true`. Correct so far. Now **step over** one line at a time. After `total += line`, `total = 81000`: still consistent, because the base price of item 3 is 45,000. The next line tests `express[i]` and enters the branch. The watch shows the surcharge about to be added: $81{,}000 \times 50/100 = 40{,}500$. But **Table 2.5** says the surcharge for item 3 is $45{,}000 \times 50/100 = 22{,}500$. One more step over, and `total` jumps to 121,500 instead of 103,500.

There it is. The surcharge is computed from `total`, the running total of *all* items so far, instead of from `line`, the price of *this* item. Items 1 and 2 were charged the express surcharge too, although they were not express: $36{,}000 \times 50/100 = 18{,}000$ too much before the discount, and $18{,}000 \times 90/100 = 16{,}200$ too much after it, which is exactly the overcharge we observed. The bug is explained completely, down to the last rupiah, which is how you know it is *the* bug and not merely *a* bug.

Written as a formula, the loop computes, for $i = 1, \ldots, n$ starting from $t_0 = 0$,

$$t_i = (t_{i-1} + p'_i)\left(1 + \frac{e_i}{2}\right), \tag{2.4}$$

where $p'_i = r(s_i)\max(w_i, 3)$ is the base price without surcharge, whereas Equation (2.2) and Equation (2.3) ask for $t_i = t_{i-1} + p'_i\left(1 + \frac{e_i}{2}\right)$. The factor $\left(1 + \frac{e_i}{2}\right)$ has been applied to the wrong thing. Equation (2.4) also tells us when the bug is invisible: if the only express item is the *first* one, $t_{i-1} = 0$ and the two formulas agree. A test with a single express item would never catch it, which is why the bug survived.

Here is the hunt in `jdb`, with the breakpoint on line 27; after three stops we step over the express branch and watch `total` change:

```
Initializing jdb ...
> stop at LaundryPrice:27
Deferring breakpoint LaundryPrice:27.
It will be set after the class is loaded.
> run
run LaundryPrice
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint LaundryPrice:27

Breakpoint hit: "thread=main", LaundryPrice.orderTotal(), line=27 bci=26
27                total += line;

main[1] locals
Method arguments:
services = instance of java.lang.String[3] (id=449)
kgs = instance of double[3] (id=450)
express = instance of boolean[3] (id=451)
member = true
Local variables:
total = 0
i = 0
line = 24000
main[1] cont
>
Breakpoint hit: "thread=main", LaundryPrice.orderTotal(), line=27 bci=26
27                total += line;

main[1] print line
 line = 12000
main[1] print total
 total = 24000
main[1] cont
>
Breakpoint hit: "thread=main", LaundryPrice.orderTotal(), line=27 bci=26
27                total += line;

main[1] print i
 i = 2
main[1] print express[i]
 express[i] = true
main[1] next
>
Step completed: "thread=main", LaundryPrice.orderTotal(), line=28 bci=33
28                if (express[i]) {
```

```
main[1] print total
 total = 81000
main[1] next
>
Step completed: "thread=main", LaundryPrice.orderTotal(), line=29 bci=40
29                    total += total * EXPRESS_PERCENT / 100;

main[1] print total
 total = 81000
main[1] next
>
Step completed: "thread=main", LaundryPrice.orderTotal(), line=25 bci=55
25            for (int i = 0; i < services.length; i++) {

main[1] print total
 total = 121500
main[1] where
  [1] LaundryPrice.orderTotal (LaundryPrice.java:25)
  [2] LaundryPrice.main (LaundryPrice.java:42)
main[1] cont
> Customer : Bu Sari (member)
Total    : Rp 109350
Expected : Rp 93150

The application exited
```

`locals` at the first stop shows everything at once: the four parameters (arrays appear as `instance of double[3]` with an internal id) and the locals `total`, `i` and `line`. At the third stop, `print i` and `print express[i]` confirm that this is the express item, and three `next` commands take `total` from 81000 (after line 27) to 121500 (after line 29). `where` confirms we are in `orderTotal`, called from `main` at line 42.

And in `pdb`, where the command `display` plays the part of a watch: every time the program stops, `pdb` reprints each displayed expression *whose value has changed*, with its old value:

```
> /home/student/laundry/laundry_price.py(1)<module>()
-> """laundry_price.py: prices one Profitina Laundry order. This version has a bug."""
(Pdb) break 28
Breakpoint 1 at /home/student/laundry/laundry_price.py:28
(Pdb) continue
> /home/student/laundry/laundry_price.py(28)order_total()
-> total += line
(Pdb) display total
display total: 0
(Pdb) display line
display line: 24000
(Pdb) next
> /home/student/laundry/laundry_price.py(29)order_total()
-> if express[i]:
display total: 24000  [old: 0]
(Pdb) next
> /home/student/laundry/laundry_price.py(26)order_total()
-> for i in range(len(services)):
(Pdb) continue
> /home/student/laundry/laundry_price.py(28)order_total()
-> total += line
display line: 12000  [old: 24000]
(Pdb) next
> /home/student/laundry/laundry_price.py(29)order_total()
-> if express[i]:
display total: 36000  [old: 24000]
(Pdb) next
> /home/student/laundry/laundry_price.py(26)order_total()
-> for i in range(len(services)):
(Pdb) continue
> /home/student/laundry/laundry_price.py(28)order_total()
-> total += line
display line: 45000  [old: 12000]
(Pdb) p i, express[i]
(2, True)
(Pdb) next
> /home/student/laundry/laundry_price.py(29)order_total()
-> if express[i]:
display total: 81000  [old: 36000]
(Pdb) next
> /home/student/laundry/laundry_price.py(30)order_total()
-> total += total * EXPRESS_PERCENT // 100
(Pdb) next
> /home/student/laundry/laundry_price.py(26)order_total()
-> for i in range(len(services)):
display total: 121500  [old: 81000]
(Pdb) quit
```

The last line of the session says it all: `display total: 121500  [old: 81000]`.

2.11.6 Step 5: fix, and check

The fix is to add the surcharge to the item's own price before adding the item to the total, exactly as Equation (2.2) says. In **Listing 2.11** and **Listing 2.12** only the loop in `orderTotal` has changed: the surcharge now multiplies `line`, and `total += line` moves after it.

Listing 2.11 `LaundryPriceFixed.java`: the express surcharge now applies to the item's own price.

```java
// LaundryPriceFixed.java: prices one Profitina Laundry order. Bug fixed.
public class LaundryPriceFixed {

    static final double MIN_KG = 3.0;            // lighter loads are billed as 3 kg
    static final long EXPRESS_PERCENT = 50;      // express costs 50% more
    static final long MEMBER_DISCOUNT_PERCENT = 10;
    static final long DISCOUNT_THRESHOLD = 100_000;

    static long pricePerKg(String service) {
        return switch (service) {
            case "wash" -> 6_000;
            case "iron" -> 4_000;
            case "wash+iron" -> 9_000;
            default -> throw new IllegalArgumentException("unknown service: " + service);
        };
    }

    static long linePrice(String service, double kg) {
        double billedKg = Math.max(kg, MIN_KG);
        return (long) (billedKg * pricePerKg(service));
    }

    static long orderTotal(String[] services, double[] kgs, boolean[] express, boolean
member) {
        long total = 0;
        for (int i = 0; i < services.length; i++) {
            long line = linePrice(services[i], kgs[i]);
            if (express[i]) {
                line += line * EXPRESS_PERCENT / 100;   // surcharge on this line only
            }
            total += line;
        }
        if (member && total >= DISCOUNT_THRESHOLD) {
            total -= total * MEMBER_DISCOUNT_PERCENT / 100;
        }
        return total;
    }
```

Listing 2.11 (Continued) *LaundryPriceFixed.java: the express surcharge now applies to the item's own price.*

```java
    public static void main(String[] args) {
        String[] services = {"wash", "iron", "wash+iron"};
        double[] kgs = {4.0, 2.0, 5.0};
        boolean[] express = {false, false, true};
        long total = orderTotal(services, kgs, express, true);
        System.out.println("Customer : Bu Sari (member)");
        System.out.println("Total    : Rp " + total);
        System.out.println("Expected : Rp 93150");
    }
}
```

```
Customer : Bu Sari (member)
Total    : Rp 93150
Expected : Rp 93150
```

Listing 2.12 `laundry_price_fixed.py`: the same fix in Python.

```
"""laundry_price_fixed.py: prices one Profitina Laundry order. Bug fixed."""

MIN_KG = 3.0                      # lighter loads are billed as 3 kg
EXPRESS_PERCENT = 50              # express costs 50% more
MEMBER_DISCOUNT_PERCENT = 10
DISCOUNT_THRESHOLD = 100_000


def price_per_kg(service):
    if service == "wash":
        return 6_000
    elif service == "iron":
        return 4_000
    elif service == "wash+iron":
        return 9_000
    raise ValueError("unknown service: " + service)


def line_price(service, kg):
    billed_kg = max(kg, MIN_KG)
    return int(billed_kg * price_per_kg(service))


def order_total(services, kgs, express, member):
    total = 0
    for i in range(len(services)):
        line = line_price(services[i], kgs[i])
        if express[i]:
            line += line * EXPRESS_PERCENT // 100   # surcharge on this line only
        total += line
    if member and total >= DISCOUNT_THRESHOLD:
        total -= total * MEMBER_DISCOUNT_PERCENT // 100
    return total


def main():
    services = ["wash", "iron", "wash+iron"]
    kgs = [4.0, 2.0, 5.0]
    express = [False, False, True]
    total = order_total(services, kgs, express, True)
    print("Customer : Bu Sari (member)")
    print("Total    : Rp", total)
    print("Expected : Rp 93150")
```

Listing 2.12 (Continued) *laundry_price_fixed.py: the same fix in Python.*

```
if __name__ == "__main__":
    main()
```

```
Customer : Bu Sari (member)
Total    : Rp 93150
Expected : Rp 93150
```

Both programs now print Rp 93,150. Do not stop at the one test that failed, though. Run the fixed program under the debugger once more with the same watches and confirm that the running total passes through 24,000, 36,000 and 103,500, as **Table 2.5** says; a fix that produces the right total for the wrong reason is a bug waiting for its next customer. Chapter 6 shows how to turn such checks into automatic tests that run every time the code changes.

2.11.7 The method behind the hunt

Look back at what we did, because the same seven steps work for every silent bug you will ever meet. **Figure 2.44** draws them as a cycle.

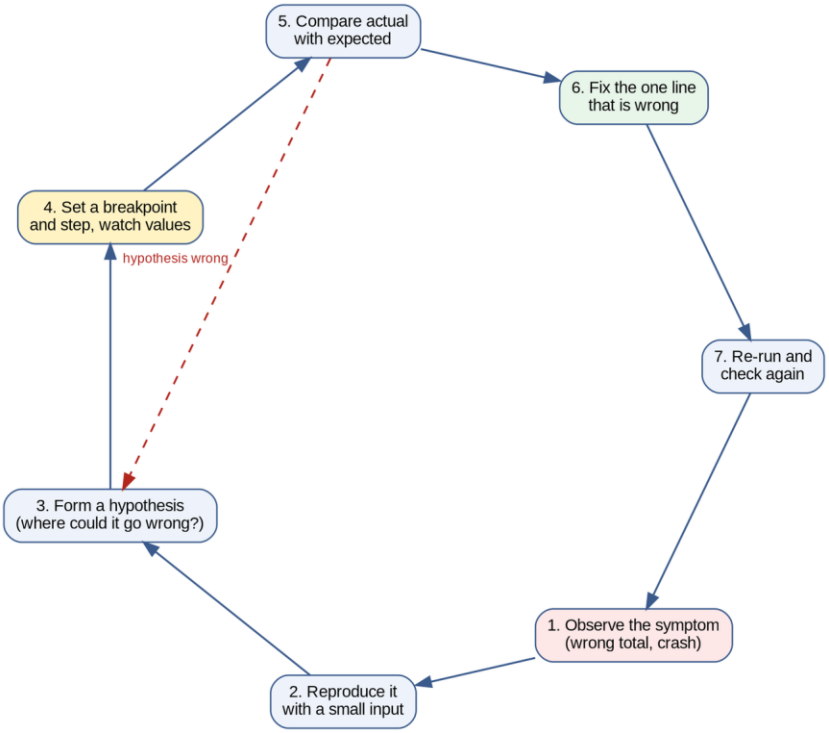


Figure 2.44 The debugging cycle. When the comparison in step 5 refutes the hypothesis, go back to step 3 with what you have learned, not back to guessing.

Observe the symptom precisely; reproduce it with the smallest input (three items here); form a hypothesis about *where* to look, preferring places where values come together; set a breakpoint and step; compare every observed value with the value worked out by hand, since the first difference is the bug; fix the one wrong line and explain the exact wrong output it caused; re-run the failing case and the cases that used to pass. If the comparison shows everything correct where you looked, the hypothesis was wrong; that is progress, because you have ruled out part of the program. Move the breakpoint and repeat. A systematic search of this kind always ends, which guessing does not [12], [13].

2.12 Conditional Breakpoints: Finding the Needle in the Haystack

A breakpoint inside a loop stops on *every* pass. With three items that is fine; with a day's orders, or a year's, stepping through every pass is hopeless. A **conditional breakpoint** stops only when an expression you choose is true.

At the end of each day Profitina Laundry adds up that day's orders. **Listing 2.13** and **Listing 2.14** do this for ten orders. (To keep the example small, these orders have no express items and no discounts.) The total is Rp 6,000 short of the figure the cashier counted in the drawer.

Listing 2.13 `DailyOrders.java`: a day's orders, Rp 6,000 short.

```java
// DailyOrders.java: adds up one day of Profitina Laundry orders. This version has a bug.
public class DailyOrders {

    static final double MIN_KG = 3.0;            // lighter loads are billed as 3 kg

    static long pricePerKg(String service) {
        return switch (service) {
            case "wash" -> 6_000;
            case "iron" -> 4_000;
            case "wash+iron" -> 9_000;
            default -> throw new IllegalArgumentException("unknown service: " + service);
        };
    }

    static long linePrice(String service, double kg) {
        long billedKg = (long) Math.max(kg, MIN_KG);
        return billedKg * pricePerKg(service);
    }

    public static void main(String[] args) {
        String[] services = {"wash", "iron", "wash+iron", "wash", "iron",
                             "wash+iron", "wash", "iron", "wash", "wash+iron"};
        double[] kgs = {3.0, 5.0, 2.0, 7.5, 4.0, 6.0, 3.5, 8.0, 1.5, 5.0};
        long day = 0;
        for (int i = 0; i < services.length; i++) {
            long price = linePrice(services[i], kgs[i]);
            System.out.printf("Order %2d  %-9s %4.1f kg  Rp %6d%n", i + 1, services[i],
kgs[i], price);
            day += price;
        }
        System.out.println("Day total : Rp " + day);
        System.out.println("Expected  : Rp 296000");
    }
}
```

```
Order  1  wash       3.0 kg  Rp  18000
Order  2  iron       5.0 kg  Rp  20000
Order  3  wash+iron  2.0 kg  Rp  27000
Order  4  wash       7.5 kg  Rp  42000
Order  5  iron       4.0 kg  Rp  16000
Order  6  wash+iron  6.0 kg  Rp  54000
Order  7  wash       3.5 kg  Rp  18000
Order  8  iron       8.0 kg  Rp  32000
Order  9  wash       1.5 kg  Rp  18000
Order 10  wash+iron  5.0 kg  Rp  45000
Day total : Rp 290000
Expected  : Rp 296000
```

Listing 2.14 `daily_orders.py`: the same day's orders in Python, with the same shortfall.

```python
"""daily_orders.py: adds up one day of Profitina Laundry orders. This version has a bug."""

MIN_KG = 3.0                      # lighter loads are billed as 3 kg


def price_per_kg(service):
    if service == "wash":
        return 6_000
    elif service == "iron":
        return 4_000
    elif service == "wash+iron":
        return 9_000
    raise ValueError("unknown service: " + service)


def line_price(service, kg):
    billed_kg = int(max(kg, MIN_KG))
    return billed_kg * price_per_kg(service)


def main():
    services = ["wash", "iron", "wash+iron", "wash", "iron",
                "wash+iron", "wash", "iron", "wash", "wash+iron"]
    kgs = [3.0, 5.0, 2.0, 7.5, 4.0, 6.0, 3.5, 8.0, 1.5, 5.0]
    day = 0
    for i in range(len(services)):
        price = line_price(services[i], kgs[i])
        print(f"Order {i + 1:2d}  {services[i]:<9} {kgs[i]:4.1f} kg  Rp {price:6d}")
        day += price
    print("Day total : Rp", day)
    print("Expected  : Rp 296000")


if __name__ == "__main__":
    main()
```

```
Order  1  wash       3.0 kg  Rp  18000
Order  2  iron       5.0 kg  Rp  20000
Order  3  wash+iron  2.0 kg  Rp  27000
Order  4  wash       7.5 kg  Rp  42000
Order  5  iron       4.0 kg  Rp  16000
Order  6  wash+iron  6.0 kg  Rp  54000
Order  7  wash       3.5 kg  Rp  18000
Order  8  iron       8.0 kg  Rp  32000
Order  9  wash       1.5 kg  Rp  18000
Order 10  wash+iron  5.0 kg  Rp  45000
Day total : Rp 290000
Expected  : Rp 296000
```

(`printf` in Java and the *f-string* in Python print columns: `%2d` is a whole number two characters wide, `%-9s` a string left-aligned in nine characters, `%4.1f` a decimal with one digit after the point; Chapter 3 explains formatting fully.)

Which order is wrong? You could put a breakpoint in `linePrice` and inspect ten stops, but think instead about what "wrong" would *look like* inside `linePrice`. The rule is that the billed weight is the larger of the real weight and 3 kg, so the billed weight must never be *less* than the real weight. A breakpoint on the `return` line of `linePrice` (Java line 17, Python line 18) with the condition

```
billedKg < kg          (Java)
billed_kg < kg         (Python)
```

will stop only for orders that break the rule.

- **Eclipse:** set the breakpoint, then right-click it and choose *Breakpoint Properties…* (or select it in the Breakpoints view); tick **Conditional**, choose *Suspend when 'true'*, and type the condition in the box below (the lower part of **Figure 2.42**).
- **VS Code:** right-click in the gutter of the line and choose *Add Conditional Breakpoint…*, select *Expression*, type the condition and press Enter. The dot gets a small equals sign.
- **pdb:** add the condition after a comma: `break 18, billed_kg < kg`.

Here is the `pdb` session:

```
> /home/student/laundry/daily_orders.py(1)<module>()
-> """daily_orders.py: adds up one day of Profitina Laundry orders. This version has a
bug."""
(Pdb) break 18, billed_kg < kg
Breakpoint 1 at /home/student/laundry/daily_orders.py:18
(Pdb) continue
Order  1  wash       3.0 kg  Rp  18000
Order  2  iron       5.0 kg  Rp  20000
Order  3  wash+iron  2.0 kg  Rp  27000
> /home/student/laundry/daily_orders.py(18)line_price()
-> return billed_kg * price_per_kg(service)
(Pdb) p service, kg, billed_kg
('wash', 7.5, 7)
(Pdb) continue
Order  4  wash       7.5 kg  Rp  42000
Order  5  iron       4.0 kg  Rp  16000
Order  6  wash+iron  6.0 kg  Rp  54000
> /home/student/laundry/daily_orders.py(18)line_price()
-> return billed_kg * price_per_kg(service)
(Pdb) p service, kg, billed_kg
('wash', 3.5, 3)
(Pdb) continue
Order  7  wash       3.5 kg  Rp  18000
Order  8  iron       8.0 kg  Rp  32000
Order  9  wash       1.5 kg  Rp  18000
Order 10  wash+iron  5.0 kg  Rp  45000
Day total : Rp 290000
Expected  : Rp 296000
The program finished and will be restarted
> /home/student/laundry/daily_orders.py(1)<module>()
-> """daily_orders.py: adds up one day of Profitina Laundry orders. This version has a
bug."""
(Pdb) quit
```

The program ran freely through orders 1 to 3 (their lines were printed) and stopped only twice: for a 7.5 kg load billed as 7 kg, and a 3.5 kg load billed as 3 kg. The shortfall is explained to the rupiah: $0.5 \times 6{,}000 + 0.5 \times 6{,}000 = 6{,}000$. The cause is the conversion to a whole number: `(long) Math.max(kg, MIN_KG)` in Java and `int(max(kg, MIN_KG))` in Python both *truncate*, that is, throw away the fraction, so half a kilogram of washing is given away free. Order 9, 1.5 kg, did *not* stop the program, because the 3 kg minimum raised it to exactly 3.0 before the truncation. The fix is to multiply first and convert the price, not the weight, exactly as `LaundryPrice` does; Exercise 2.6 asks you to make it and check the new total.

Two related tools: a **hit count** stops only on the n-th arrival (VS Code: *Hit Count*; `pdb`: `ignore 1 6` skips the next six hits of breakpoint 1), and an **exception breakpoint** stops the moment an exception is thrown, anywhere (Eclipse: the *J!* button of the Breakpoints view; VS Code: tick *Uncaught Exceptions* or *Raised Exceptions*). That brings us to the loud bugs.

2.13 Reading Stack Traces

When a Java or Python program meets a situation it cannot handle, it *throws* (Python: *raises*) an **exception**. If nothing catches it, the program stops and prints a **stack trace** (Python: a *traceback*): a message naming the exception, and the call stack at the moment it was thrown. Beginners often see a stack trace as a wall of red text and scroll past it. In fact it is the most useful message the runtime ever gives you, because it says *what* went wrong, *where*, and *how the program got there*.

Profitina Laundry has started offering dry cleaning, but nobody has added it to the price list yet. **Listing 2.15** and **Listing 2.16** price an order that includes it.

Listing 2.15 `UnknownService.java`: an order naming a service the price list does not know.

```java
// UnknownService.java: an order that names a service the price list does not know.
public class UnknownService {

    static final double MIN_KG = 3.0;

    static long pricePerKg(String service) {
        return switch (service) {
            case "wash" -> 6_000;
            case "iron" -> 4_000;
            case "wash+iron" -> 9_000;
            default -> throw new IllegalArgumentException("unknown service: " + service);
        };
    }

    static long linePrice(String service, double kg) {
        double billedKg = Math.max(kg, MIN_KG);
        return (long) (billedKg * pricePerKg(service));
    }

    public static void main(String[] args) {
        String[] services = {"wash", "dry-clean"};
        double[] kgs = {4.0, 2.0};
        for (int i = 0; i < services.length; i++) {
            System.out.println(services[i] + ": Rp " + linePrice(services[i], kgs[i]));
        }
    }
}
```

```
wash: Rp 24000
Exception in thread "main" java.lang.IllegalArgumentException: unknown service: dry-clean
    at UnknownService.pricePerKg(UnknownService.java:11)
    at UnknownService.linePrice(UnknownService.java:17)
    at UnknownService.main(UnknownService.java:24)
```

Listing 2.16 `unknown_service.py`: the same order in Python.

```
"""unknown_service.py: an order that names a service the price list does not know."""

MIN_KG = 3.0


def price_per_kg(service):
    if service == "wash":
        return 6_000
    elif service == "iron":
        return 4_000
    elif service == "wash+iron":
        return 9_000
    raise ValueError("unknown service: " + service)


def line_price(service, kg):
    billed_kg = max(kg, MIN_KG)
    return int(billed_kg * price_per_kg(service))


def main():
    services = ["wash", "dry-clean"]
    kgs = [4.0, 2.0]
    for i in range(len(services)):
        print(services[i] + ": Rp", line_price(services[i], kgs[i]))


if __name__ == "__main__":
    main()
```

```
wash: Rp 24000
Traceback (most recent call last):
  File "/home/student/laundry/unknown_service.py", line 29, in <module>
    main()
  File "/home/student/laundry/unknown_service.py", line 25, in main
    print(services[i] + ": Rp", line_price(services[i], kgs[i]))
                               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/student/laundry/unknown_service.py", line 18, in line_price
    return int(billed_kg * price_per_kg(service))
                           ^^^^^^^^^^^^^^^^^^^^^
  File "/home/student/laundry/unknown_service.py", line 13, in price_per_kg
    raise ValueError("unknown service: " + service)
ValueError: unknown service: dry-clean
```

2.13.1 Reading the Java stack trace

Read a Java stack trace from the top.

1. `Exception in thread "main"`: the exception was not caught anywhere, in the thread that runs `main`.
2. `java.lang.IllegalArgumentException`: the *type* of exception, with its package. The name alone often tells the story: an argument was illegal.
3. `unknown service: dry-clean`: the *message*, the text given when the exception was created. This is why a good exception message includes the offending value.
4. The lines starting with `at` are the call stack, **most recent call first**. The top one, `UnknownService.pricePerKg(UnknownService.java:11)`, is where the exception was thrown: method `pricePerKg`, file `UnknownService.java`, line 11 (the `default` branch). Below it, `linePrice` at line 17 called `pricePerKg`, and `main` at line 24 called `linePrice`.

The line that *threw* is not always the line that is *wrong*: `pricePerKg` rightly refused an unknown service, and the real question is why `main` asked for one. In real programs, skip the library frames

(`java.base/...`) and look for the frames that name your own classes. In both IDEs each `(File.java:11)` in the Console is a link to that line.

2.13.2 Reading the Python traceback

Python prints the same information in the *opposite order*, as its first line warns: `Traceback (most recent call last)`. So read a Python traceback **from the bottom**.

1. The last line, `ValueError: unknown service: dry-clean`, gives the exception type and message.
2. Just above it, the innermost frame: `File ".../unknown_service.py", line 13, in price_per_kg`, followed by the source line itself, `raise ValueError(...)`.
3. Above that, `line_price` at line 18 called it, and above that `main` at line 25, and at the top the module-level call `main()` at line 29.

Python 3.11 and later add lines of carets (`^^^^`) under the exact part of a line that was being evaluated, which is very helpful when a line contains several calls: here the carets under `line_price(services[i], kgs[i])` and `price_per_kg(service)` point to the calls themselves, not to the `print` or the `int` around them. (Python 3.13 and 3.14 also print tracebacks in color in the terminal.) The path shown for each file is the full path on the machine where the program ran, so it will be different on yours.

The first line, `wash: Rp 24000`, shows that the program did useful work before failing. Stack traces go to the *standard error* stream, so a terminal mixes them with the output, but redirecting the output to a file saves only the normal lines.

> **KEY INSIGHT.** A stack trace is the call stack of Section 2.10.5, printed at the moment of the crash. Java lists it newest-first and Python oldest-first, but they contain the same three facts: the exception type, its message, and the chain of calls with file names and line numbers. Read all three before you change a single line.

The debugger can take you one step further. With an exception breakpoint (Section 2.12) on `IllegalArgumentException` in Eclipse or VS Code, or with *Raised Exceptions* ticked for Python, the program stops at the `throw` or `raise` line *before* the stack unwinds, with every variable of every frame still available to inspect. `pdb` does the same automatically in *post-mortem* mode: `python3 -m pdb unknown_service.py`, then `continue`, stops at the line that raised the exception so that you can type `where` and `p service`.

2.14 Profitina in Practice

Profitina, the AI Visibility platform at `app.profitina.com`, is a Python web application built with FastAPI and served by Uvicorn, with PostgreSQL in production and dependencies pinned in a lock file. Three habits from this chapter are visible in how such a product is kept healthy.

Reproduce before you fix. When a brand owner reports that a visibility score “looks wrong”, the first step is not to read code but to reproduce the scan on a test copy with the same input, as in step 2 of **Figure 2.44**. Only a bug that can be reproduced can be stepped through with breakpoints, and only a fixed bug that is then turned into an automated test stays fixed. Profitina treats its full automated test suite, well over a thousand tests, as a release gate: nothing is deployed until all of them pass.

Stack traces are data. A web server cannot stop at a breakpoint in front of real users, so when a request fails, the server records the traceback in its log instead of showing it to the user. Reading

those tracebacks bottom-up, exactly as in Section 2.13.2, is how a failing request is traced to the frame where the decision went wrong. Tracebacks are recorded without any personal data, in line with the Indonesian personal data protection law (UU PDP 27/2022).

Know which machine runs what. Profitina's AI features run locally on Profitina's own server, with Qwen2.5 served through Ollama, so no user data leaves the server for AI processing; when no model is available the app says so instead of pretending. Debugging such a system starts with the question of Section 2.7.3 in a larger form: which interpreter, which model, which configuration is actually running? A health endpoint that reports the deployed version, and a release process that succeeds only after that endpoint answers, are the production equivalent of looking at the status bar before pressing F5.

As always, the app advises and the user decides; debugging tools help developers make the advice correct, never make it a guarantee.

2.15 Common Mistakes, and How to Avoid Them

Running the old program. You edited the file but did not save it (look for `*` on the tab), or in a terminal you edited `LaundryPrice.java` and ran `java LaundryPrice` without running `javac` again. *Avoid:* save before running; in a terminal, compile and run together, or use `java LaundryPrice.java`.

The wrong Python. VS Code runs a different interpreter from the one where you installed a package, so imports fail. *Avoid:* check the interpreter in the status bar, and read the full interpreter path VS Code types into the Terminal.

Running a package module as a script. `python3 main/tester.py` fails with `ModuleNotFoundError: No module named 'util'`. *Avoid:* go to the folder that contains the packages and run `python3 -m main.tester`.

Package and folder disagree. A Java file that says `package util;` but lives in `src/tools/` does not compile in Eclipse and cannot be found by `java`. *Avoid:* create classes through *New > Class* in the right package, and move them with *Refactor > Move*, which updates the folder and every import.

Pressing Run instead of Debug, or naming the class wrongly. Breakpoints are ignored under *Run As*, and `java main/Tester.class` fails. *Avoid:* use *Debug As* or F5 in VS Code, and name classes with dots: `java -cp bin main.Tester`.

Stepping into everything. Stepping into every call takes you deep into code you trust. *Avoid:* step over what you trust, step into what you suspect, step out as soon as you are lost.

Stopping at the first wrong value without explaining it. Fixing the first suspicious line without checking that it explains the whole symptom often fixes *a* bug but not *the* bug. *Avoid:* do the arithmetic, as in Section 2.11.5, until the fix explains the error to the last rupiah.

Reading the stack trace from the wrong end. *Avoid:* Java lists the newest call first, Python last; find the exception type and message, then the first frame that is your own code.

Leaving debug prints behind. *Avoid:* prefer watches and logpoints, and search for leftover prints before you finish.

2.16 Chapter Summary

- An **IDE** integrates editing, building, running, debugging and navigation. **Table 2.2** translates between Eclipse and VS Code.
- The CPU runs only **machine code**. Java compiles source to **bytecode** once with `javac`, and the **JVM** interprets and JIT-compiles that bytecode each run. Python compiles source to bytecode inside the interpreter each time a program starts, caching imported modules in `__pycache__`, and its virtual machine runs the bytecode.
- Java finds missing names and wrong types at **compile time**; Python finds syntax errors before running but missing names only at **run time**, when the line executes.
- Eclipse has a **workspace** of **projects** and a window of **views** and **editors** arranged in **perspectives**; VS Code works on a **folder** and runs Python with the interpreter selected in the status bar.
- A **package** is a folder of related code; **imports** bring names from other packages. Java classes run as `java -cp bin package.Class`; Python modules in packages run as `python -m package.module` from the folder above the packages.
- A **debugger** suspends the program at **breakpoints** and lets you **step over**, **step into** and **step out**, inspect and change **variables**, keep **watches**, and read the **call stack**. **Conditional** and **hit-count** breakpoints stop only when it matters; **exception breakpoints** stop where an exception is thrown.
- Debug by method: observe, reproduce, hypothesize, set a breakpoint, compare actual with expected values worked out by hand, fix, re-run (**Figure 2.44**). The laundry bill was too high because the express surcharge multiplied the running total instead of the item price (Equation (2.4)).
- A **stack trace** gives the exception type, its message and the chain of calls; read Java's from the top and Python's from the bottom.

2.17 Exercises

Exercise 2.1. In one sentence each, say what an Eclipse *workspace*, a *project*, a *package*, a *view*, an *editor* and a *perspective* are, and name the nearest VS Code equivalent of each.

Exercise 2.2. For each mistake, say whether Java reports it at compile time or run time, and whether Python reports it before the program starts or while it runs: (a) a missing `;` in Java or a missing `:` in Python; (b) a misspelled variable name inside an `if` branch that runs only on Sundays; (c) calling `count(300)` on the `Counter` of **Listing 2.5** and **Listing 2.7**; (d) a missing import of `Counter`.

Exercise 2.3. A file `src/com/profitina/laundry/Receipt.java` begins with `package com.profitina.laundry;` and contains `main`. Write the commands that compile it into a folder `bin` and run it, and say where `Receipt.class` ends up.

Exercise 2.4. In the `python` folder of Section 2.8.3, a student types `python3 main/tester.py` and gets `ModuleNotFoundError: No module named 'util'`. Explain why, and give the correct command.

Exercise 2.5. The program **Listing 2.11** is suspended at its first stop on line 26. Starting there, you press Step Into three times, then Step Out twice. After each of the five presses, give the method and line where the program is suspended and the number of frames of your own code on the call stack.

Exercise 2.6. Fix **Listing 2.13** and **Listing 2.14** so that the day total is Rp 296,000. Which conditional breakpoint shows that the fix works, and what does it do after the fix?

Exercise 2.7. A tester calls `count(5)` twice on the same `Counter` of **Listing 2.5** and gets 15 and then 30. Use the debugger (which variable would you watch, and where would you put the breakpoint?) to explain why, and write a version whose second call also returns 15.

Exercise 2.8. Profitina Laundry adds dry cleaning at Rp 15,000 per kg, with the same 3 kg minimum. Write a program in both languages that prints an itemized receipt for a member's order of 4 kg wash, 2 kg dry-clean express, and 3.5 kg iron, followed by the subtotal, the member discount and the total. Work the total out by hand first.

Exercise 2.9. Using Equation (2.4), compute what the *buggy* **Listing 2.9** would charge Bu Sari if her three items were entered in the order wash+iron (express), wash, iron. Explain what this tells you about testing.

Exercise 2.10. The buggy program of **Listing 2.9** is suspended at its third stop on line 27, and you use the Variables view to change `express[2]` from `true` to `false`, then resume. What total does the program print, and why is this a useful experiment?

Exercise 2.11. Explain the difference between a breakpoint, a conditional breakpoint, a hit-count breakpoint, an exception breakpoint, a watchpoint and a VS Code logpoint, and give one laundry situation in which each is the best choice.

2.18 Solutions to the Exercises

Solution 2.1. A *workspace* is the folder holding Eclipse's metadata and, by default, your projects (VS Code: the opened folder or a `.code-workspace` file). A *project* is a buildable unit with a `.project` file (VS Code: a project shown in the Java Projects view). A *package* is a named group of classes stored as a folder (the same in both). A *view* shows structured information such as the Package Explorer (VS Code: side-bar views and panels). An *editor* holds one file and must be saved (VS Code: editor tabs). A *perspective* arranges views for a task (VS Code: one layout that adds the Run and Debug views when debugging).

Solution 2.2. (a) Java: compile time; Python: before the program starts (`SyntaxError`), because Python compiles the whole file first. (b) Java: compile time (`cannot find symbol`), even though the branch rarely runs; Python: at run time, and only on a Sunday, when the branch runs (`NameError`). (c) Both at run time: the value 300 is only known when the call happens, and the method throws. (d) Java: compile time; Python: at run time, when the name is first used (Section 2.8.4).

Solution 2.3. From the folder that contains `src`: `javac -d bin src/com/profitina/laundry/Receipt.java`, then `java -cp bin com.profitina.laundry.Receipt`. The class file is `bin/com/profitina/laundry/Receipt.class`, because `-d` creates one folder per package level.

Solution 2.4. Started as a script path, Python searches for imports in the script's own folder, `python/main`, which has no `util` inside it. Run it as a module from the folder that contains both packages: `cd python`, then `python3 -m main.tester`.

Solution 2.5. Starting in `orderTotal` at line 26 (two frames: `orderTotal`, `main`). (1) Step Into: `linePrice`, line 19, three frames. (2) Step Into: `linePrice`, line 20, three frames; `Math.max` is skipped by the step filter. (3) Step Into: `pricePerKg`, line 10, four frames. (4) Step Out: back in

`linePrice`, line 20, three frames, still completing the multiplication. (5) Step Out: back in `orderTotal`, line 26, two frames; one Step Over then completes the assignment `line = 24000`. This is the path of the `jdb` tour in Section 2.11.4, whose line numbers are the same because the fixed program changes only lines 27 to 30.

Solution 2.6. Convert the *price*, not the weight. In Java, replace the two lines of `linePrice` by `double billedKg = Math.max(kg, MIN_KG);` and `return (long) (billedKg * pricePerKg(service));`; in Python, by `billed_kg = max(kg, MIN_KG)` and `return int(billed_kg * price_per_kg(service))`. Orders 4 and 7 become Rp 45,000 and Rp 21,000, and the day total becomes 290,000 + 3,000 + 3,000 = 296,000. The same conditional breakpoint, `billedKg < kg` on the `return` line, now never stops: the program runs straight to the end, which confirms that no order is billed below its real weight.

Solution 2.7. Put a breakpoint on the line `result += i;` (line 15 of **Listing 2.5**) and watch `result` and `i`. On the second call the first stop already shows `result = 15`: `result` is a *field*, which belongs to the object and survives between calls, not a local variable that starts again at 0. The fix is to add up in a local variable, as `countFresh` does in **Listing 2.17** and **Listing 2.18**, which show both behaviors side by side.

Listing 2.17 `CounterTwice.java`: a field keeps its value between calls; a local variable starts fresh.

```
// CounterTwice.java: why a second call to count(5) returns 30, and the fix.
public class CounterTwice {

    private int result = 0;              // a field: it survives between calls

    int countKeepingTotal(int x) {       // the original behavior
        for (int i = 1; i <= x; i++) {
            result += i;
        }
        return result;
    }

    int countFresh(int x) {              // the fix: a local variable, new on every call
        int sum = 0;
        for (int i = 1; i <= x; i++) {
            sum += i;
        }
        return sum;
    }

    public static void main(String[] args) {
        CounterTwice c = new CounterTwice();
        System.out.println("original, 1st call: " + c.countKeepingTotal(5));
        System.out.println("original, 2nd call: " + c.countKeepingTotal(5));
        System.out.println("fixed,    1st call: " + c.countFresh(5));
        System.out.println("fixed,    2nd call: " + c.countFresh(5));
    }
}
```

```
original, 1st call: 15
original, 2nd call: 30
fixed,    1st call: 15
fixed,    2nd call: 15
```

Listing 2.18 `counter_twice.py`: the same contrast in Python, with an attribute and a local variable.

```
"""counter_twice.py: why a second call to count(5) returns 30, and the fix."""


class CounterTwice:

    def __init__(self):
        self.result = 0                  # an attribute: it survives between calls

    def count_keeping_total(self, x):    # the original behavior
        for i in range(1, x + 1):
            self.result += i
        return self.result

    def count_fresh(self, x):            # the fix: a local variable, new on every call
        total = 0
        for i in range(1, x + 1):
            total += i
        return total


def main():
    c = CounterTwice()
    print("original, 1st call:", c.count_keeping_total(5))
    print("original, 2nd call:", c.count_keeping_total(5))
    print("fixed,    1st call:", c.count_fresh(5))
    print("fixed,    2nd call:", c.count_fresh(5))


if __name__ == "__main__":
    main()
```

```
original, 1st call: 15
original, 2nd call: 30
fixed,    1st call: 15
fixed,    2nd call: 15
```

Solution 2.8. By hand, with Equation (2.2): wash $6{,}000 \times 4 = 24{,}000$; dry-clean $15{,}000 \times \max(2{,}3) = 45{,}000$, plus 50% express, gives 67,500; iron $4{,}000 \times 3.5 = 14{,}000$. The subtotal is $105{,}500 \geq 100{,}000$, so the member discount is 10,550 and, by Equation (2.3), the total is 94,950. **Listing 2.19** and **Listing 2.20** compute the surcharge inside `linePrice`, where it belongs, so the bug of Section 2.11 cannot come back.

Listing 2.19 `LaundryReceipt.java`: an itemized receipt with dry cleaning added.

```java
// LaundryReceipt.java: an itemized receipt, with the dry-clean service added.
public class LaundryReceipt {

    static final double MIN_KG = 3.0;            // lighter loads are billed as 3 kg
    static final long EXPRESS_PERCENT = 50;      // express costs 50% more
    static final long MEMBER_DISCOUNT_PERCENT = 10;
    static final long DISCOUNT_THRESHOLD = 100_000;

    static long pricePerKg(String service) {
        return switch (service) {
            case "wash" -> 6_000;
            case "iron" -> 4_000;
            case "wash+iron" -> 9_000;
            case "dry-clean" -> 15_000;
            default -> throw new IllegalArgumentException("unknown service: " + service);
        };
    }

    static long linePrice(String service, double kg, boolean express) {
        double billedKg = Math.max(kg, MIN_KG);
        long line = (long) (billedKg * pricePerKg(service));
        if (express) {
            line += line * EXPRESS_PERCENT / 100;
        }
        return line;
    }

    public static void main(String[] args) {
        String[] services = {"wash", "dry-clean", "iron"};
        double[] kgs = {4.0, 2.0, 3.5};
        boolean[] express = {false, true, false};
        boolean member = true;
        long subtotal = 0;
        for (int i = 0; i < services.length; i++) {
            long line = linePrice(services[i], kgs[i], express[i]);
            System.out.printf("%-9s %4.1f kg %-7s Rp %7d%n",
                    services[i], kgs[i], express[i] ? "express" : "", line);
            subtotal += line;
        }
        long discount = 0;
        if (member && subtotal >= DISCOUNT_THRESHOLD) {
            discount = subtotal * MEMBER_DISCOUNT_PERCENT / 100;
        }
        System.out.printf("Subtotal                 Rp %7d%n", subtotal);
        System.out.printf("Member discount          Rp %7d%n", -discount);
        System.out.printf("Total                    Rp %7d%n", subtotal - discount);
```

Listing 2.19 (Continued) *LaundryReceipt.java: an itemized receipt with dry cleaning added.*

```java
    }
}
```

```
wash       4.0 kg         Rp   24000
dry-clean  2.0 kg express Rp   67500
iron       3.5 kg         Rp   14000
Subtotal                  Rp  105500
Member discount           Rp  -10550
Total                     Rp   94950
```

Listing 2.20 `laundry_receipt.py`: the same receipt in Python; the `%` formatting uses the same codes as Java's `printf`.

```
"""laundry_receipt.py: an itemized receipt, with the dry-clean service added."""

MIN_KG = 3.0                    # lighter loads are billed as 3 kg
EXPRESS_PERCENT = 50            # express costs 50% more
MEMBER_DISCOUNT_PERCENT = 10
DISCOUNT_THRESHOLD = 100_000


def price_per_kg(service):
    if service == "wash":
        return 6_000
    elif service == "iron":
        return 4_000
    elif service == "wash+iron":
        return 9_000
    elif service == "dry-clean":
        return 15_000
    raise ValueError("unknown service: " + service)


def line_price(service, kg, express):
    billed_kg = max(kg, MIN_KG)
    line = int(billed_kg * price_per_kg(service))
    if express:
        line += line * EXPRESS_PERCENT // 100
    return line


def main():
    services = ["wash", "dry-clean", "iron"]
    kgs = [4.0, 2.0, 3.5]
    express = [False, True, False]
    member = True
    subtotal = 0
    for i in range(len(services)):
        line = line_price(services[i], kgs[i], express[i])
        print("%-9s %4.1f kg %-7s Rp %7d"
              % (services[i], kgs[i], "express" if express[i] else "", line))
        subtotal += line
    discount = 0
    if member and subtotal >= DISCOUNT_THRESHOLD:
        discount = subtotal * MEMBER_DISCOUNT_PERCENT // 100
    print("Subtotal                  Rp %7d" % subtotal)
    print("Member discount           Rp %7d" % -discount)
    print("Total                     Rp %7d" % (subtotal - discount))
```

Listing 2.20 (Continued) *laundry_receipt.py: the same receipt in Python; the % formatting uses the same codes as Java's printf.*

```
if __name__ == "__main__":
    main()
```

```
wash       4.0 kg         Rp   24000
dry-clean  2.0 kg express Rp   67500
iron       3.5 kg         Rp   14000
Subtotal                  Rp  105500
Member discount           Rp  -10550
Total                     Rp   94950
```

Solution 2.9. With $t_0 = 0$: $t_1 = (0 + 45{,}000) \times 1.5 = 67{,}500$; $t_2 = 67{,}500 + 24{,}000 = 91{,}500$; $t_3 = 91{,}500 + 12{,}000 = 103{,}500$. After the discount the bill is 93,150, the *correct* amount. The bug is invisible whenever the only express item comes first, so a test must include an express item after other items. A test suite should be designed to make bugs visible, not merely to run the code.

Solution 2.10. With the express flag switched off, the branch is skipped, so `total` stays 81,000; since 81,000 < 100,000, no discount applies and the program prints `Total     : Rp 81000`. That agrees with Equation (2.2) and Equation (2.3) for three non-express items (24,000 + 12,000 + 45,000), which shows that everything except the express branch is correct. Changing a value in the debugger answers such a "what if" without editing or recompiling the program.

Solution 2.11. A *breakpoint* stops every time a line is reached (the line that adds an item to a bill). A *conditional breakpoint* stops when an expression is true (`billedKg < kg` in Section 2.12). A *hit-count breakpoint* stops on the n-th arrival (the 200th order of a month). An *exception breakpoint* stops where an exception is thrown (the unknown service `dry-clean`, with all variables intact). A *watchpoint* stops when a field is read or written (which method changes a customer's balance?). A *logpoint* prints without stopping (every running total of a day's orders).

References

[1] Eclipse Foundation, "Eclipse IDE documentation: Java development user guide," 2025. [Online]. Available: https://help.eclipse.org/

[2] L. Vogel, "Eclipse IDE for Java development: tutorial," vogella GmbH, 2024. [Online]. Available: https://www.vogella.com/tutorials/Eclipse/article.html

[3] Microsoft, "Java in Visual Studio Code" and "Running and debugging Java," Visual Studio Code documentation, 2025. [Online]. Available: https://code.visualstudio.com/docs/java/java-tutorial

[4] Microsoft, "Python in Visual Studio Code" and "Python debugging in VS Code," Visual Studio Code documentation, 2025. [Online]. Available: https://code.visualstudio.com/docs/python/python-tutorial

[5] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, *The Java Virtual Machine Specification, Java SE 21 Edition*. Redwood City, CA, USA: Oracle, 2023.

[6] Oracle, *Java Development Kit Version 21 Tool Specifications* (javac, java, jar, javap, jdb), 2023. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/specs/man/

[7] Python Software Foundation, "dis — Disassembler for Python bytecode," *The Python Standard Library*, version 3.14, 2025. [Online]. Available: https://docs.python.org/3/library/dis.html

[8] Python Software Foundation, "Modules," *The Python Tutorial*, sec. 6, version 3.14, 2025. [Online]. Available: https://docs.python.org/3/tutorial/modules.html

[9] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*, ch. 7, "Packages and Modules." Redwood City, CA, USA: Oracle, 2023.

[10] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2018.

[11] Python Software Foundation, "pdb — The Python Debugger," *The Python Standard Library*, version 3.14, 2025. [Online]. Available: https://docs.python.org/3/library/pdb.html

[12] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*, 2nd ed. Burlington, MA, USA: Morgan Kaufmann, 2009.

[13] D. J. Agans, *Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*. New York, NY, USA: AMACOM, 2002.

Chapter 3

Classes, Objects, Strings, Types, Conditionals, and Loops

By the end of this chapter you will be able to read any small Java or Python class and name every part of it: fields, constructor, methods, and the word `this` (or `self`) that ties them together. You will create objects and draw them in memory, handle text confidently, choose the right number types and recognize the three arithmetic traps, make decisions with `if` and `switch`, repeat work with loops, and check a loop by hand with a tracing table. Finally, you will build the first two classes of Profitina Laundry, `Customer` and `Service`, which compute the price of a bag of laundry, and run every program in both Java and Python.

3.1 Where We Left Off

Chapter 1 set up your workbench, a Java Development Kit, a Python interpreter and Visual Studio Code, and you ran a first program in each language. Chapter 2 made the workbench comfortable: projects and packages, Eclipse and VS Code side by side, and the debugger, with its breakpoints, stepping and watched variables. The programs themselves, though, have been tiny. This chapter opens a program up and lays every part out on the workbench.

The parts fall into two groups. The first is about *structure*: **classes** and **objects**. A class describes a kind of thing: a dog, a customer, a laundry service. An object is one particular thing of that kind: the dog named Fred, the customer named Siti. Object-oriented programming is the craft of choosing good classes and letting objects work together.

The second group is about *what happens inside a method*: the **values** a program works with, the **decisions** it makes and the **repetition** it performs. A programmer who has not mastered them writes beautiful classes that compute the wrong answer. At the end, Profitina Laundry, the small business whose software we build chapter by chapter, gets its first two classes.

> **KEY INSIGHT.** A class is a *description*; an object is a *thing*. Everything else in this chapter (strings, types, conditionals, loops) is what goes on inside the methods that give objects their behavior.

3.2 A First Look Inside a Class

3.2.1 Hello, once more

The simplest thing a program can do is print a line of text. You met this program in Chapter 1; let us now read it slowly, word by word. **Figure 3.1** shows it open in the Eclipse editor, and **Listing 3.1** gives the same text as a listing you can copy.

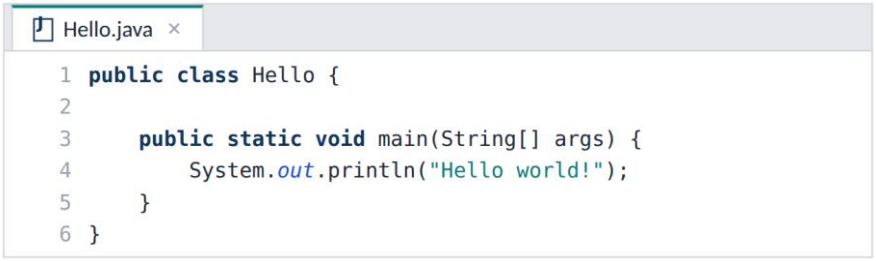


Figure 3.1 The class `Hello` open in the Eclipse editor: one class, one `main` method, one statement.

Listing 3.1 `Hello.java`: the smallest complete Java program.

```
public class Hello {

    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

```
Hello world!
```

Every word in **Listing 3.1** earns its place.

- `public class Hello` declares a **class** named `Hello`. In Java, *all* code lives inside some class. The word `public` means that any other code may use this class. The file must be called `Hello.java`, exactly matching the class name, capital `H` included.
- The curly braces `{` and `}` mark where the class begins and ends. Everything between them belongs to `Hello`.
- `public static void main(String[] args)` declares a **method** called `main`, a named block of instructions. The Java launcher starts the program at a method of exactly this shape when you type `java Hello`. We explain `static` properly in a later chapter; for now read it as "this method belongs to the class itself, so Java can call it before any object exists". `void` means the method gives back no value, and `String[] args` holds the words typed after the program name, which we use in a moment.
- `System.out.println("Hello world!");` is a **statement**: one instruction. It asks the object `System.out` (the standard output, normally your terminal window) to print a line.

Notice what is fussy. The capital `S` in `System`, the dots, the double quotes and the closing semicolon are all required; get any of them wrong and the compiler refuses the program. That precision is what lets Java catch mistakes before your program runs.

Here is the same program in Python, in **Listing 3.2**.

Listing 3.2 `hello.py`: the same program in Python, with the conventional `main` function.

```
def main():
    print("Hello world!")


if __name__ == "__main__":
    main()
```

```
Hello world!
```

Python needs no class for a program this small: the single line `print("Hello world!")` would work. We wrap it in a function `main` and call it under `if __name__ == "__main__":`, which is true only when the file is run directly, not when another file imports it. Later in this chapter we import classes from one file into another, and then the habit pays off.

Two differences between the languages show up already. Java marks blocks with braces; Python marks them with **indentation**: the body of `main` is the lines indented under `def main():`. And Java ends statements with semicolons, while Python ends them at the end of the line.

3.2.2 Reading words from the command line

A program that can only write is half a program. The simplest way to give a program some input is to type words after its name when you start it. Java collects those words into `args`. **Figure 3.2** shows a second version of the program, `Hello2`, that uses them.



Figure 3.2 `Hello2` in Eclipse: the program reads the first command-line word from `args[0]` if one was given.

Listing 3.3 `Hello2.java`: printing the first word typed on the command line.

```
public class Hello2 {

    public static void main(String[] args) {
        String aLineOfText = "";
        if (args.length > 0) {
            aLineOfText = args[0];
            System.out.println("She said, " + aLineOfText + "!");
        } else {
            System.out.println("Hello world!");
        }
    }
}
```

If you run it with the words `Welcome to Profitina Laundry` after the class name,

```
java Hello2 Welcome to Profitina Laundry
```

then `args` holds four words: `args[0]` is `Welcome`, `args[1]` is `to`, `args[2]` is `Profitina` and `args[3]` is `Laundry`. The program prints only the first:

```
She said, Welcome!
```

Run it with no words at all and `args.length` is `0`, so the `else` branch runs and the program falls back to its greeting:

```
Hello world!
```

Three things in **Listing 3.3** deserve a closer look.

First, **numbering starts at 0**. The first word is `args[0]`, not `args[1]`. Almost everything in computing counts from zero: positions in a list, characters in a string, rows in a table. It feels strange for a week and then natural.

Second, `String aLineOfText = "";` declares a **variable**, a named box that holds a value. Its type is `String`, which is Java's type for text. `String` is an *object type*: a `String` value is an object with methods of its own, as we will see in Section 3.5. For now it is enough to know that we store the text of `args[0]` in the variable `aLineOfText` and then print it.

Third, the `+` sign glues text together, which is called **concatenation**: `"She said, " + aLineOfText + "!"` builds one string from three.

The test `if (args.length > 0)` protects us. Without it, running the program with no words would ask for a nonexistent `args[0]`, and Java would stop with an `ArrayIndexOutOfBoundsException`, a kind of error Chapter 5 studies properly.

The Python version, **Listing 3.4**, reads the same words from `sys.argv`. The one difference to remember is that `sys.argv[0]` is the name of the script itself, so the first real word is `sys.argv[1]`.

Listing 3.4 `hello2.py`: the command-line words live in `sys.argv`, after the script name.

```
import sys


def main():
    a_line_of_text = ""
    if len(sys.argv) > 1:
        a_line_of_text = sys.argv[1]
        print("She said, " + a_line_of_text + "!")
    else:
        print("Hello world!")


if __name__ == "__main__":
    main()
```

```
She said, Welcome!
```

3.2.3 Names and conventions

Programs are read far more often than they are written, so every language community agrees on how names should look. The compiler does not enforce these conventions, but every experienced reader expects them.

Java. Class names start with a capital letter: `Hello`, `Dog`, `LaundryDemo`. A public class `MyClass` must live in a file named `MyClass.java`. Variables and methods start with a lowercase letter: `aLineOfText`, `getAge`, `myIntValue`. When a name is made of several words, each new word starts with a capital letter and there are no spaces or underscores: `myIntValue`, `kilogramsPerPound`. This style is called **camel case**, because the capitals stick up like the humps of a camel. Constants that never change are written in capitals with underscores: `MAX_VALUE`.

Python. Class names also use capitals, `Dog`, `LaundryService` (the style is sometimes called *CapWords*). Variables and functions are written in lowercase with underscores between words: `a_line_of_text`, `get_age`, `kilograms_per_pound`. This style is called **snake case**. The official style guide for Python code, PEP 8 [1], describes these rules and many more.

Some things are not conventions but rules. Java's own names must be spelled with the right capitals: `System` and `String` start with capitals, `int` and `char` do not. If you write `string s = "...";` the compiler reports that it cannot find a type called `string`. Python is just as strict: `print` is a function, `Print` is an unknown name.

3.2.4 Comments that help the reader

A **comment** is text in a program that the computer ignores and the human reader does not. Comments explain *why* the code does what it does, which the code itself cannot say. The habit pays off the first time you open a program you wrote three months ago.

Java has three kinds, and Python has two, all shown in **Listing 3.5** and **Listing 3.6**.

Listing 3.5 `CommentDemo.java`: a Javadoc comment, a line comment and a block comment.

```
/**
 * Prints a short receipt line for one laundry order.
 * A Javadoc comment like this one describes a class or a method
 * for the people who will use it.
 */
public class CommentDemo {

    public static void main(String[] args) {
        String customer = "Budi Santoso";
        String shorterName = customer.substring(0, 4); // first four characters, for the
tag

        /* The tag printed on the laundry bag must fit in a small box,
         * so we use the short form of the name followed by the weight.
         */
        double weightKg = 3.5;
        System.out.println("Bag tag: " + shorterName + " / " + weightKg + " kg");
    }
}
```

```
Bag tag: Budi / 3.5 kg
```

Listing 3.6 `comment_demo.py`: a docstring and line comments in Python.

```
"""Prints a short receipt line for one laundry order.

A docstring like this one describes a module, a class or a function
for the people who will use it.
"""


def main():
    customer = "Budi Santoso"
    shorter_name = customer[0:4]  # first four characters, for the tag

    # The tag printed on the laundry bag must fit in a small box,
    # so we use the short form of the name followed by the weight.
    weight_kg = 3.5
    print("Bag tag: " + shorter_name + " / " + str(weight_kg) + " kg")


if __name__ == "__main__":
    main()
```

```
Bag tag: Budi / 3.5 kg
```

- A **line comment** starts with `//` in Java and `#` in Python and runs to the end of the line. It suits a short remark about one line, such as "first four characters, for the tag".

- A **block comment** in Java starts with `/*` and ends with `*/`, and may span many lines. It usually sits *before* the code it explains. Python has no block comment; you simply write several `#` lines.
- A **documentation comment** describes a class or method for the people who will *use* it rather than read its inside. In Java it starts with `/**` and is called a *Javadoc* comment; the `javadoc` tool turns these into web pages. In Python it is a string placed as the first statement of a module, class or function, called a **docstring**. Chapter 6 treats both in depth.

The hard part of commenting is judgment. Compare:

```
int i;                                  // Declare an int called i
i = 3;                                  // Set i to 3
System.out.println("i is equal to " + i); // Print what i is equal to
```

Each comment repeats the code in English and adds nothing. A good rule: *if a classmate could easily understand what a line does, do not comment it*. Comment the business rule and the reason: “members get 10% off” is a good comment, because the code shows *that* a tenth is subtracted but not *why*.

3.3 Classes and Objects

3.3.1 Blueprints and houses

Think of a **class** as a blueprint. An architect’s blueprint for a house says that the house has a front door, three windows and a kitchen of a certain size, but you cannot live in a blueprint. From one blueprint a builder can build many houses, each a real building you can walk into. Each house can be painted a different color, and each has its own furniture. An **object** is one of those houses: a real thing, built from the class, holding its own data [2].

Take a dog. A class `Dog` describes what every dog in our program has: a name, a breed, an age and a weight. But nobody owns “a generic dog”: every real dog is a particular one, such as Fred, a three-year-old Yorkshire Terrier weighing 7.6 kg, or Bella, a four-year-old Rhodesian Ridgeback weighing 11.6 kg. Fred and Bella are two **instances** of the class `Dog`; *object* and *instance* mean the same thing.

A second picture helps many readers: a **spreadsheet**. The class is like the row of column headers, *name*, *breed*, *age*, *weight*. Each object is like one row of values beneath them. The headers say what can be recorded; the rows are the records.

Table 3.1 A class seen as a spreadsheet: the header row is the class `Dog`, each row below it is one `Dog` object.

Name	Breed	Age	Weight
Fred	Yorkshire Terrier	3	7.6
Bella	Rhodesian Ridgeback	4	11.6

Table 3.1 shows the spreadsheet picture, and **Figure 3.3** shows the same idea in the drawing style programmers use every day, the **UML class diagram** on the left and two **object diagrams** on the right. The class box lists the fields (with a `-` sign, meaning private) and the methods (with a `+`, meaning public). Each object box has an underlined title, *variable name : class name*, and lists the current value of every field.

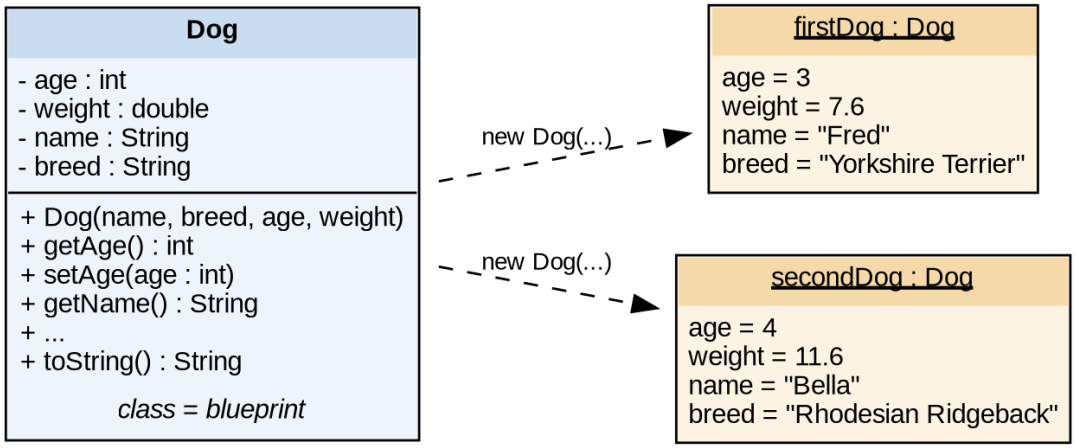


Figure 3.3 The class `Dog` as a blueprint (left) and two objects built from it with `new` (right); each object has its own copy of every field.

The most important thing in **Figure 3.3** is easy to miss: *each object has its own copy of every field*. Fred's `age` and Bella's `age` are two different boxes in memory. Changing one leaves the other alone.

3.3.2 The anatomy of a class

Now let us write the class. **Listing 3.7** is the complete `Dog` in Java.

Listing 3.7 `Dog.java`: four private fields, one constructor, a getter and a setter for each field, and `toString`.

```
public class Dog {
    // Fields: the data every Dog object carries
    private int age;
    private double weight;
    private String name;
    private String breed;

    /* No main method here: Dog is not a program on its own.
     * It only needs a constructor, which builds a new Dog.
     */
    public Dog(String name, String breed, int age, double weight) {
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.breed = breed;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public double getWeight() {
        return weight;
    }

    public void setWeight(double weight) {
        this.weight = weight;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getBreed() {
        return breed;
    }
```

Listing 3.7 (Continued) *Dog.java: four private fields, one constructor, a getter and a setter for each field, and toString.*

```
    public void setBreed(String breed) {
        this.breed = breed;
    }

    @Override
    public String toString() {
        return "Dog name: " + getName() + "; breed: " + getBreed()
                + "; age: " + getAge() + "; weight: " + getWeight() + " kg";
    }
}
```

The same class, as it looks in the Eclipse editor, appears in **Figure 3.4** and **Figure 3.5**; the line numbers in the margin make it easy to point at parts while you read.

Dog.java × DogTest.java

```
1  public class Dog {
2      // Fields: the data every Dog object carries
3      private int age;
4      private double weight;
5      private String name;
6      private String breed;
7
8      /* No main method here: Dog is not a program on its own.
9       * It only needs a constructor, which builds a new Dog.
10      */
11     public Dog(String name, String breed, int age, double weight) {
12         this.name = name;
13         this.age = age;
14         this.weight = weight;
15         this.breed = breed;
16     }
17
18     public int getAge() {
19         return age;
20     }
21
22     public void setAge(int age) {
23         this.age = age;
24     }
25
26     public double getWeight() {
27         return weight;
28     }
29
30     public void setWeight(double weight) {
31         this.weight = weight;
32     }
33
```

Figure 3.4 The first part of `Dog.java` in Eclipse: the fields, the constructor and the first getters and setters.

Dog.java × DogTest.java

```
34     public String getName() {
35         return name;
36     }
37
38     public void setName(String name) {
39         this.name = name;
40     }
41
42     public String getBreed() {
43         return breed;
44     }
45
46     public void setBreed(String breed) {
47         this.breed = breed;
48     }
49
50     @Override
51     public String toString() {
52         return "Dog name: " + getName() + "; breed: " + getBreed()
53                 + "; age: " + getAge() + "; weight: " + getWeight() + " kg";
54     }
55 }
```

Figure 3.5 The rest of `Dog.java` in Eclipse: the remaining getters and setters and `toString`.

A class body contains three kinds of **members**. Learn their names now; every later chapter uses them.

Fields (also called *instance variables* or *attributes*) are variables declared inside the class but outside any method. They hold the object's data, `int age`, `double weight`, and each object gets its own copy. The word `private` means only code inside `Dog` may touch the field directly, so the class can keep its data sensible, for example by refusing a negative age. Hiding data behind methods is called **encapsulation**, the first of the great ideas of object-oriented programming; Chapter 7 studies access control fully.

A **constructor** is a special method that builds a new object and gives its fields their first values. You can recognize it because it has the same name as the class and no return type, not even `void`: `public Dog(String name, String breed, int age, double weight)`. The four names in parentheses are its **parameters**: variables that receive the values the caller supplies. Notice that `Dog` has no `main` method. It is not a program; it is a blueprint that other programs use.

Methods are the object's behavior. `getAge()` hands back the dog's age; `setAge(int age)` changes it; `toString()` describes the whole dog as text. A method declaration states what the method returns (`int`, `String`, or `void` for nothing), its name, and its parameters.

Table 3.2 lines up the Java parts with their Python counterparts.

Table 3.2 The parts of a class in Java and in Python.

Part	Java	Python
Class header	`public class Dog {`	`class Dog:`
Field	`private int age;` declared in the class body	`self._age = age` assigned inside `__init__`
Constructor	`public Dog(String name, ...)`	`def __init__(self, name, ...):`
The current object	`this` (usually optional)	`self` (always written, first parameter)
Getter	`public int getAge() { return age; }`	`def get_age(self): return self._age`
Setter	`public void setAge(int age) { this.age = age; }`	`def set_age(self, age): self._age = age`
Text form	`public String toString()`	`def __str__(self):`
Private	keyword `private`, enforced by the compiler	leading underscore `_age`, a convention

Here is the Python class, **Listing 3.8**.

Listing 3.8 `dog.py`: the class `Dog` in Python; fields are created inside `__init__`.

```
class Dog:
    """A dog with a name, a breed, an age in years and a weight in kg."""

    def __init__(self, name, breed, age, weight):
        # The leading underscore says: "private, please use the methods"
        self._name = name
        self._age = age
        self._weight = weight
        self._breed = breed

    def get_age(self):
        return self._age

    def set_age(self, age):
        self._age = age

    def get_weight(self):
        return self._weight

    def set_weight(self, weight):
        self._weight = weight

    def get_name(self):
        return self._name

    def set_name(self, name):
        self._name = name

    def get_breed(self):
        return self._breed

    def set_breed(self, breed):
        self._breed = breed

    def __str__(self):
        return ("Dog name: " + self.get_name() + "; breed: " + self.get_breed()
                + "; age: " + str(self.get_age()) + "; weight: "
                + str(self.get_weight()) + " kg")
```

Read it against **Table 3.2** and the differences are small but real.

- Python does not declare fields in the class body: `self._age = age` in the constructor creates the field `_age` on this object.
- The constructor is always called `__init__`: Python creates the empty object first, and `__init__` *initializes* it.
- Python has no `private` keyword. A leading underscore, `_age`, tells other programmers "this is internal; please use the methods"; Python does not enforce it.
- Python writes no types in this class. A Python variable can hold a value of any type; we return to this in Section 3.6.2.

3.3.3 The word `this`, and the word `self`

Look again at the constructor in **Listing 3.7**:

```
public Dog(String name, String breed, int age, double weight) {
    this.name = name;
    ...
}
```

There are now *two* things called `name`: the field `name` that belongs to the object, and the parameter `name` that holds the value the caller passed in. Inside the constructor, a plain `name` means the nearest one, the parameter. So what would `name = name;` do? It would copy the parameter into

itself and leave the field untouched, and the new dog would have no name (its `name` field would stay `null`). The compiler accepts it without complaint; it is a silent bug.

The keyword `this` solves the puzzle. `this` means "the object this code is currently working on", so `this.name = name;` reads "set the `name` field *of this object* to the parameter `name`". Where there is no clash, as in `getAge()`, which says `return age;`, you may leave `this` out.

Python makes the idea explicit and compulsory. Every method receives the current object as its first parameter, conventionally called `self`, and you must always write `self._age` to reach a field. A plain `_age` inside a method would be a local variable, not the field. When you write `first_dog.get_age()`, Python quietly calls `Dog.get_age(first_dog)`: the object before the dot becomes `self`.

> **DEFINITION.** Inside a method or constructor, `this` (Java) or `self` (Python) refers to the object on which the method was called, or, in a constructor, the object being built. `this.x` or `self.x` is the field `x` of that object.

3.3.4 Getters, setters, and `toString`

Because the fields are private, the outside world needs a polite way in. That is what **getters** and **setters** are for.

A **getter** returns the value of one field: `public String getName() { return name; }`. The keyword `return` ends the method and hands the value back to whoever called it. A method whose return type is anything other than `void` must end in a `return` statement on every path; the compiler checks this.

A **setter** changes the value of one field: `public void setName(String name) { this.name = name; }`. Its return type is `void`, because it has nothing to hand back, and so it needs no `return`.

Why not make the fields public? Because the setter is a gate. Tomorrow you may add one line to `setAge`, `if (age < 0) return;`, and every caller is protected at once; a public field has no gate to add. A class may also simply offer no setter, as `Customer` in Section 3.12 does for data that must never change.

`toString` returns a text description of the object. Every Java object inherits one from the class `Object` (Chapter 7), but it prints something like `Dog@1b6d3586`, which looks like nonsense; writing our own decides what a dog looks like when printed. `@Override` tells the compiler "I intend to replace an inherited method", so a misspelled `tostring` is reported as an error. Our `toString` calls the getters rather than reading the fields, so that any later work a getter does (formatting, computing) benefits `toString` automatically.

In Python the same job is done by `__str__`. Whenever Python needs an object as text, in `print(dog)` or `str(dog)`, it calls `__str__`.

3.3.5 Creating objects with `new`

A class on its own does nothing. To use it we must **create** (or **instantiate**) objects. In Java the way to do that is the keyword `new`:

```
Dog fido = new Dog("Fido", "Jack Russell", 5, 12.5);
```

Three things happen in that one line. First, `new` asks Java to find room in memory for a new `Dog` object, with space for its four fields. Second, Java calls the constructor, passing the four **arguments** `"Fido"`, `"Jack Russell"`, `5` and `12.5`; the constructor copies them into the fields.

Third, `new` hands back a **reference** to the new object, a kind of arrow pointing at it, and the assignment stores that arrow in the variable `fido`.

The arguments must match the parameters in number, order and type: `new Dog("Fido", 5)` does not compile, and neither does `new Dog("Fido", "Jack Russell", 12.5, 5)`, because `12.5` cannot go into the `int` parameter `age`.

In Python there is no `new` keyword; you call the class as if it were a function, `Dog("Fido", "Jack Russell", 5, 12.5)`, and Python creates the object and runs `__init__` on it.

Once an object exists, we call its methods with a dot: `fido.getAge()`, `fido.setWeight(13.0)`. **Listing 3.9** puts two dogs through their paces.

Listing 3.9 `DogDemo.java`: creating two `Dog` objects, printing them, and changing them through setters.

```
public class DogDemo {

    public static void main(String[] args) {
        Dog firstDog = new Dog("Fred", "Yorkshire Terrier", 3, 7.6);
        Dog secondDog = new Dog("Bella", "Rhodesian Ridgeback", 4, 11.6);

        System.out.println(firstDog);
        System.out.println(secondDog.toString()); // same effect as the line above

        firstDog.setAge(6);
        secondDog.setWeight(14.0);
        System.out.println(secondDog.getName() + " is now "
                + secondDog.getWeight() + " kg"); // should be 14.0
        System.out.println(firstDog.getName() + " is now "
                + firstDog.getAge() + " years old"); // should be 6
    }
}
```

To compile and run, both files must be compiled together, because `DogDemo` uses `Dog`:

```
javac Dog.java DogDemo.java
java DogDemo
```

```
Dog name: Fred; breed: Yorkshire Terrier; age: 3; weight: 7.6 kg
Dog name: Bella; breed: Rhodesian Ridgeback; age: 4; weight: 11.6 kg
Bella is now 14.0 kg
Fred is now 6 years old
```

Read the output against the code. `println(firstDog)` needs text, so it calls `firstDog.toString()` for us; the second line calls it explicitly, with the same effect. The setters then change Fred's age and Bella's weight, and the getters confirm it. Fred's change does not touch Bella: each object has its own fields.

The Python driver in **Listing 3.10** imports the class from `dog.py` with `from dog import Dog`. For the import to work, both files must be in the same folder.

Listing 3.10 `dog_demo.py`: the same demonstration in Python.

```python
from dog import Dog


def main():
    first_dog = Dog("Fred", "Yorkshire Terrier", 3, 7.6)
    second_dog = Dog("Bella", "Rhodesian Ridgeback", 4, 11.6)

    print(first_dog)
    print(str(second_dog))  # same effect as the line above

    first_dog.set_age(6)
    second_dog.set_weight(14.0)
    print(second_dog.get_name() + " is now "
          + str(second_dog.get_weight()) + " kg")  # should be 14.0
    print(first_dog.get_name() + " is now "
          + str(first_dog.get_age()) + " years old")  # should be 6


if __name__ == "__main__":
    main()
```

```
Dog name: Fred; breed: Yorkshire Terrier; age: 3; weight: 7.6 kg
Dog name: Bella; breed: Rhodesian Ridgeback; age: 4; weight: 11.6 kg
Bella is now 14.0 kg
Fred is now 6 years old
```

One difference needs comment. Java's `+` turns a number into text when the other side is a string; Python demands an explicit `str(...)`. Section 3.6.2 explains why.

3.3.6 Checking that a class works

How do we know `Dog` is correct? By creating objects, calling methods, and checking that the values that come back are the values we expect. Programmers call this **testing**, and it is the difference between hoping and knowing.

The simplest possible test is an ordinary program that compares expected and actual values and says PASS or FAIL. **Listing 3.11** checks that `getAge()` returns the age given to the constructor, and that a weight set with `setWeight` is the weight read back; **Listing 3.12** is its Python twin.

Listing 3.11 `DogCheck.java`: a hand-made test of `getAge` and `setWeight`.

```java
public class DogCheck {

    public static void main(String[] args) {
        Dog firstDog = new Dog("Fred", "Yorkshire Terrier", 3, 7.6);
        int expected = 3;                  // what getAge() should give back
        int actual = firstDog.getAge();    // what it really gives back

        if (expected == actual) {
            System.out.println("PASS: getAge() returned " + actual);
        } else {
            System.out.println("FAIL: expected " + expected + " but got " + actual);
        }

        firstDog.setWeight(8.1);
        double expectedWeight = 8.1;
        if (Math.abs(firstDog.getWeight() - expectedWeight) < 1e-9) {
            System.out.println("PASS: setWeight() then getWeight() gave " +
firstDog.getWeight());
        } else {
            System.out.println("FAIL: weight is " + firstDog.getWeight());
        }
    }
}
```

```
PASS: getAge() returned 3
PASS: setWeight() then getWeight() gave 8.1
```

Listing 3.12 `dog_check.py`: the same check in Python.

```python
from dog import Dog


def main():
    first_dog = Dog("Fred", "Yorkshire Terrier", 3, 7.6)
    expected = 3                   # what get_age() should give back
    actual = first_dog.get_age()   # what it really gives back

    if expected == actual:
        print("PASS: get_age() returned " + str(actual))
    else:
        print("FAIL: expected " + str(expected) + " but got " + str(actual))

    first_dog.set_weight(8.1)
    expected_weight = 8.1
    if abs(first_dog.get_weight() - expected_weight) < 1e-9:
        print("PASS: set_weight() then get_weight() gave "
              + str(first_dog.get_weight()))
    else:
        print("FAIL: weight is " + str(first_dog.get_weight()))


if __name__ == "__main__":
    main()
```

```
PASS: get_age() returned 3
PASS: set_weight() then get_weight() gave 8.1
```

Notice that the weight is not compared with `==` but by asking whether the difference is tiny. Section 3.7.3 explains why comparing two decimal numbers with `==` is a trap.

Writing such checks by hand soon gets tedious, so professional programmers use a **testing framework**, a library that runs many small tests and reports the results; in Java the standard one is **JUnit**, whose current version is JUnit 6 [3]. **Figure 3.6** shows a JUnit test class, `DogTest`, for the

same `getAge` check, and **Figure 3.7** shows it in an Eclipse project called `dog`, next to `Dog.java`, with the JUnit library file added to the project.

Dog.java | DogTest.java ×

```
1  import static org.junit.jupiter.api.Assertions.assertEquals;
2
3  import org.junit.jupiter.api.Test;
4
5  class DogTest {
6
7      /**
8       * Testing the getAge() method.
9       */
10     @Test
11     void test1() {
12         Dog firstDog = new Dog("Fred", "Yorkshire Terrier", 3, 7.6);
13         // 3 is what I expect if getAge() is called
14         int expected = 3;
15         // The actual value
16         int firstDogAge = firstDog.getAge();
17         // By JUnit convention the expected value comes first;
18         // the message is shown only if the assertion fails
19         assertEquals(expected, firstDogAge, "test1 of getAge() in DogTest");
20     }
21 }
```

Figure 3.6 A JUnit test class `DogTest` in Eclipse; the method marked `@Test` creates a dog and uses `assertEquals` to compare the expected age with the actual one.

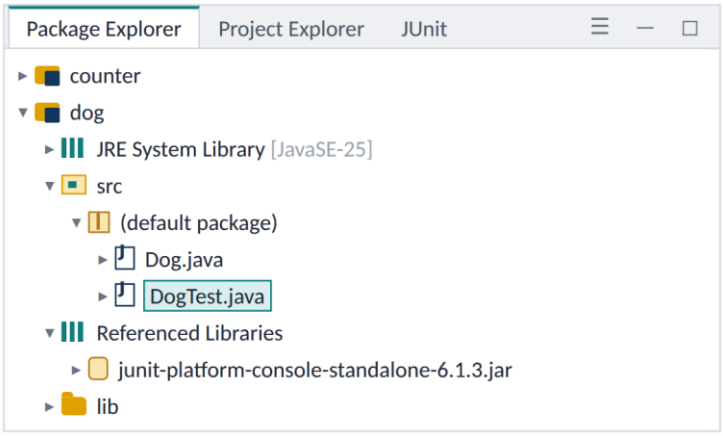


Figure 3.7 The Eclipse Package Explorer: `Dog.java` and `DogTest.java` in the project `dog`, with the JUnit library file under *Referenced Libraries*.

The test creates Fred and calls `assertEquals(expected, firstDogAge, message)`, expected value first by JUnit convention; if the two differ, the test fails and JUnit reports the message together with both values.

Eclipse reports the result with a colored bar. **Figure 3.8** shows a red bar, and the failure is instructive. In this version of `Dog` the constructor says `age = age;` instead of `this.age = age;`, so the parameter `age` is assigned to itself and the field keeps its default value 0; Eclipse even warns that the assignment has no effect. The test expected 3, the object answered 0, and the *Failure Trace* says exactly that: `expected: <3> but was: <0>`, together with the message and the line of the assertion. Once `this.` is restored, the bar turns green, as in **Figure 3.9**: one test run, no

errors, no failures. A test that turns a silent bug into a red bar with the two values side by side has done its job.

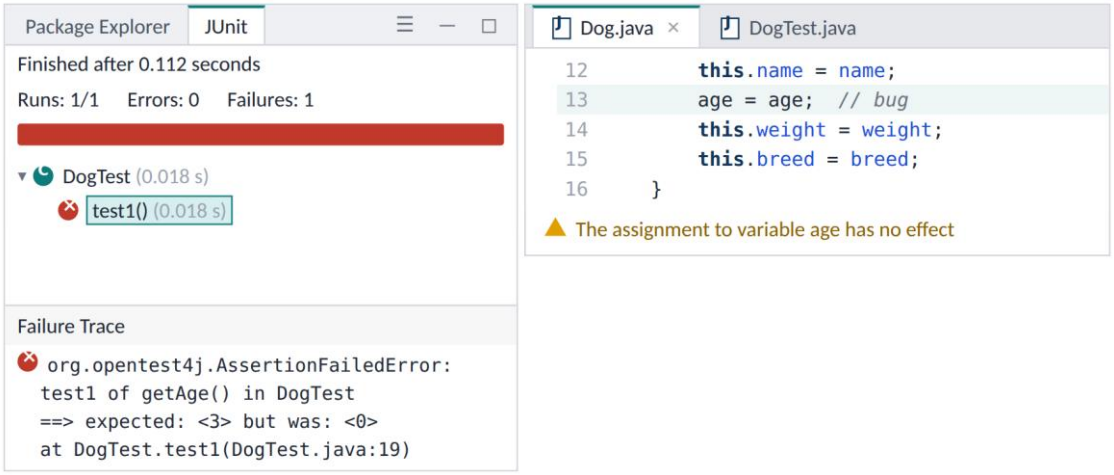


Figure 3.8 A red bar: `test1` fails with expected 3 but was 0, because the constructor of this `Dog` says `age = age;` instead of `this.age = age;`.

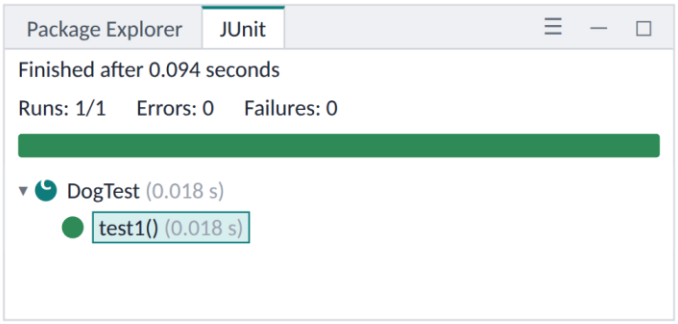


Figure 3.9 A green bar: after the fix, the test runs and passes.

Chapter 6 sets up JUnit 6 and Python's `unittest` properly and makes testing a daily habit. For now, remember the shape of every test: *arrange* an object, *act* on it by calling a method, and *assert* that the result is what you expected.

3.4 References and Memory Pictures

3.4.1 Values and arrows

Here is a question that separates programmers who understand objects from those who only use them. After these lines, what is Fido's age?

```
Dog fido = new Dog("Fido", "Jack Russell", 5, 12.5);
Dog alias = fido;
alias.setAge(6);
```

If you answered "still 5, because we changed `alias`, not `fido`", you are in good company, and you are wrong. The answer is 6. To see why, we need a picture of memory.

Java has two families of types. The **primitive types**, such as `int`, `double`, `char` and `boolean`, are simple values. A variable of a primitive type is a box that holds the value itself. The **reference types** are everything else: `String`, `Dog`, every class you will ever write. A variable of a reference type does *not* hold the object. It holds a **reference**, an arrow pointing to where the object lives.

The objects themselves live in a region of memory called the **heap**; the variables of a running method live in a **stack frame**.

That difference changes what `=` does. For primitives, `int b = a;` copies the *value*: two separate boxes now hold the same number, and changing one leaves the other alone. For references, `Dog alias = fido;` copies the *arrow*: two arrows now point at the *same* object, and a setter called through either changes the one object both see. Two variables that refer to the same object are **aliases**.

Listing 3.13 demonstrates all of this, and **Figure 3.10** draws memory just before the last two statements.

Listing 3.13 `ReferenceDemo.java`: copying a primitive copies the value; copying a reference copies the arrow.

```
public class ReferenceDemo {

    public static void main(String[] args) {
        // A primitive variable holds the value itself.
        int a = 5;
        int b = a;          // b gets a COPY of the value 5
        b = 9;
        System.out.println("a = " + a + ", b = " + b);

        // A reference variable holds an arrow to an object.
        Dog fido = new Dog("Fido", "Jack Russell", 5, 12.5);
        Dog alias = fido;   // alias gets a copy of the ARROW, not of the dog
        alias.setAge(6);
        System.out.println("fido's age seen through fido: " + fido.getAge());

        Dog twin = new Dog("Fido", "Jack Russell", 6, 12.5);
        System.out.println("fido == alias: " + (fido == alias));
        System.out.println("fido == twin:  " + (fido == twin));

        alias = null;       // alias now points nowhere; fido is untouched
        System.out.println("alias is null: " + (alias == null));
        System.out.println("fido still: " + fido);
    }
}
```

```
a = 5, b = 9
fido's age seen through fido: 6
fido == alias: true
fido == twin:  false
alias is null: true
fido still: Dog name: Fido; breed: Jack Russell; age: 6; weight: 12.5 kg
```

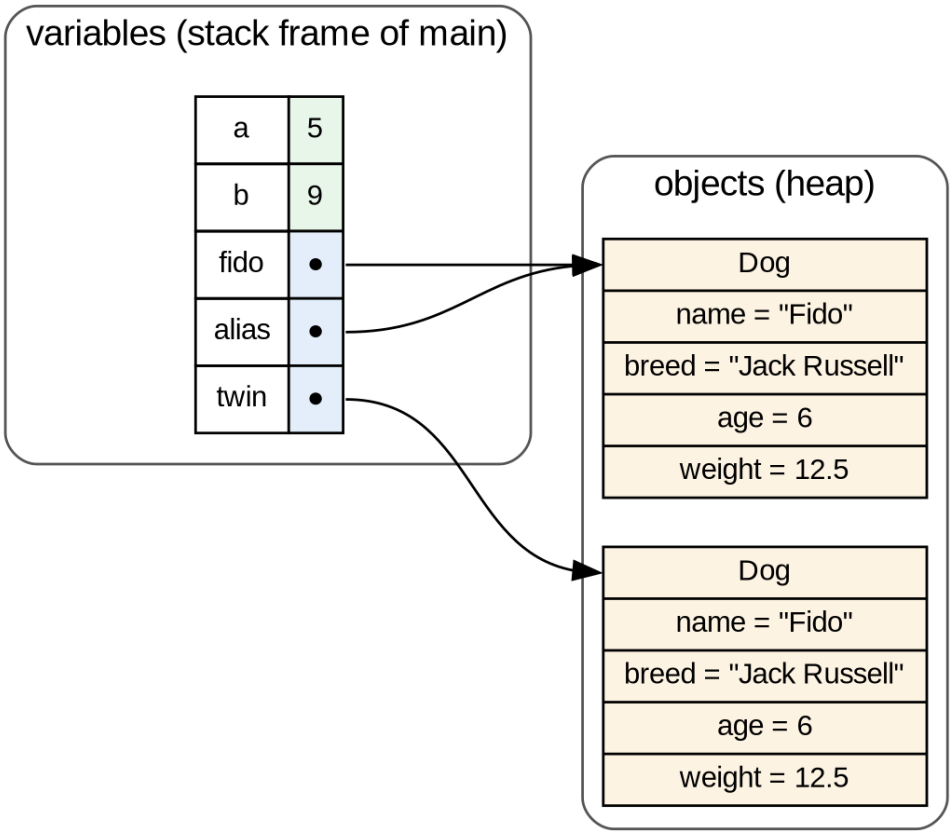


Figure 3.10 Memory inside `main` of **Listing 3.13**: `a` and `b` hold values, while `fido`, `alias` and `twin` hold arrows; `fido` and `alias` point to the same object.

Walk through the output with the picture in front of you.

1. `a = 5, b = 9`: `b` got a copy of 5 and was then overwritten with 9; `a` is untouched.
2. `fido's age seen through fido: 6`: we changed the age through `alias`, but `alias` and `fido` point to the same dog.
3. `fido == alias: true`: for references, `==` asks "do these two arrows point to the *same* object?" They do.
4. `fido == twin: false`: `twin` is a *different* object with the same field values. `==` compares arrows, not contents. Two objects with equal contents are *equal*; two arrows to one object are *identical*.
5. `alias = null` makes `alias` point to nothing; `null` means "no object". The dog is unaffected, and `fido` still reaches it.

What happens to an object that no arrow points to any more? Nothing you need to do: the Java runtime notices from time to time that the object is unreachable and reclaims its memory. This is called **garbage collection**. Python does the same.

> **WARNING.** Calling a method through a `null` reference, for example `alias.getAge()` after `alias = null`, stops the program with a `NullPointerException`. It is the most common runtime error in Java. Before calling a method on a reference that might be `null`, check it: `if (alias != null) { ... }`.

3.4.2 Python: every variable is a name

Python simplifies the picture by having only one family: *every* value is an object, including the integer 5, and *every* variable is a name bound to an object. Assignment never copies an object; it binds a name to an object. **Listing 3.14** is the Python twin of the Java demonstration.

Listing 3.14 `reference_demo.py`: in Python every variable is a name bound to an object.

```
from dog import Dog


def main():
    # An int is an immutable object: "changing" b binds b to another object.
    a = 5
    b = a
    b = 9
    print("a = " + str(a) + ", b = " + str(b))

    # Every Python variable is a name bound to an object.
    fido = Dog("Fido", "Jack Russell", 5, 12.5)
    alias = fido  # alias is a second name for the SAME dog
    alias.set_age(6)
    print("fido's age seen through fido: " + str(fido.get_age()))

    twin = Dog("Fido", "Jack Russell", 6, 12.5)
    print("fido is alias: " + str(fido is alias))
    print("fido is twin:  " + str(fido is twin))

    alias = None  # alias now names nothing; fido is untouched
    print("alias is None: " + str(alias is None))
    print("fido still: " + str(fido))


if __name__ == "__main__":
    main()
```

```
a = 5, b = 9
fido's age seen through fido: 6
fido is alias: True
fido is twin:  False
alias is None: True
fido still: Dog name: Fido; breed: Jack Russell; age: 6; weight: 12.5 kg
```

The results agree with Java, and **Figure 3.11** shows why. `b = a` binds the name `b` to the same integer object as `a`; `b = 9` then *rebinds* `b` to another object. The object 5 is unchanged, because Python integers are **immutable**: no operation changes the value inside one; a name can only be moved to a different object. The dog is **mutable**: `set_age` changes it in place, and every name bound to it sees the change.

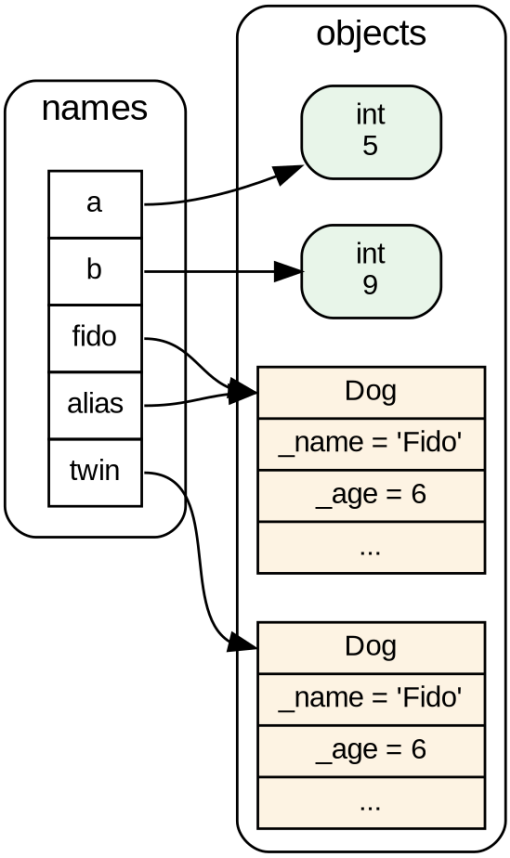


Figure 3.11 Names and objects in **Listing 3.14**: every name is an arrow; `fido` and `alias` are bound to the same `Dog` object.

Python spells identity as `is` and "no object" as `None`. The operator `==` in Python asks about equality of *content* (for classes that define what equal content means), which is the question Java asks with the method `equals`. We will see this matters a great deal for strings.

Table 3.3 Identity, equality, and "no object" in the two languages.

Question	Java	Python
Same object? (identity)	`a == b` for references	`a is b`
Same content? (equality)	`a.equals(b)`	`a == b`
No object	`null`	`None`
Test for no object	`x == null`	`x is None`

Table 3.3 is worth copying onto a card: the two languages spell the two questions in opposite ways.

3.4.3 How to draw an object diagram

Drawing memory is the most effective way to understand a program that uses objects, and it needs only a pencil. These are the rules behind **Figure 3.10**.

1. Draw one box for every variable in the method you are tracing, labeled with the variable's name. Stack them on the left.
2. For a primitive variable (`int`, `double`, `boolean`, `char`), write the value inside the box.

3. For a reference variable, draw a dot inside the box and an arrow from the dot to an object, or write `null` if it points nowhere.
4. Draw each object once, on the right, as a box with the class name on top and one line per field, *field = value*. A field that is itself a reference (such as `name`, a `String`) may be drawn with its value in quotes for brevity.
5. When a statement runs, update the picture: `new` adds an object; `x = y` for references moves `x`'s arrow to wherever `y`'s arrow points; a setter changes a value inside an object box.
6. An object with no arrows pointing to it is garbage; cross it out.

Do this for every program in this chapter at least once, and "why did my object change?" will stop surprising you.

3.5 Strings

3.5.1 Text is an object

A **string** is a sequence of characters: a word, a sentence, a name, a phone number. In Java a string has the type `String`, and it is an *object* type, which is why `String` begins with a capital letter. Underneath, a string is built from a simpler, primitive type, `char`, which holds exactly one character. The two are written differently:

```
String myString = "Hello, my name is Yarik";
char myCharacter = 'y';
```

Strings go in double quotes; single characters go in single quotes. `"y"` is a `String` of length one; `'y'` is a `char`. They are different types and cannot be swapped freely.

Python has no separate character type. Its type `str` covers text of any length, and a "character" is simply a string of length one. Python accepts both quote styles, `'y'` and `"y"`, and they mean exactly the same thing.

Much of real programming is text: names, codes, receipts, data typed by humans. The Java class `String` has dozens of methods, described in the Java API documentation [4], and Python's `str` is just as rich [5]. Here are the ones you will use every week.

3.5.2 Positions and the everyday methods

The characters of a string are numbered from 0. In `"abcde"`, the character at position 0 is `a`, and the character at position 3 is `d`. The number of characters is the string's **length**, here 5, so the last position is always `length - 1`.

Listing 3.15 exercises the everyday methods, starting with the examples from Section 3.2.

Listing 3.15 `StringBasics.java`: `startsWith`, `substring`, `charAt`, `length` and other everyday `String` methods.

```
public class StringBasics {

    public static void main(String[] args) {
        String myString = "the quick brown fox jumps over the lazy dog";
        if (myString.startsWith("the")) {
            System.out.println("It does!");
        } else {
            System.out.println("It doesn't!");
        }

        String mySecondString = "HelloYarik";
        String rest = mySecondString.substring(5);         // from index 5 to the end
        String firstWord = mySecondString.substring(0, 5); // indices 0,1,2,3,4
        System.out.println(rest + " / " + firstWord);

        String anotherString = "abcde";
        char characterThree = anotherString.charAt(3);
        System.out.println("charAt(3) = " + characterThree);
        System.out.println("length() = " + anotherString.length());

        String service = "  Express Wash  ";
        System.out.println("[" + service.trim() + "]");
        System.out.println(service.trim().toUpperCase());
        System.out.println(service.trim().toLowerCase());
        System.out.println("indexOf(\"Wash\") = " + service.trim().indexOf("Wash"));
        System.out.println("contains(\"press\") = " + service.contains("press"));
        System.out.println(service.trim().replace("Wash", "Iron"));
        System.out.println("endsWith(\"dog\") = " + myString.endsWith("dog"));

        String[] words = myString.split(" ");
        System.out.println("number of words = " + words.length);
        System.out.println("equalsIgnoreCase: " + "WASH".equalsIgnoreCase("wash"));
    }
}
```

```
It does!
Yarik / Hello
charAt(3) = d
length() = 5
[Express Wash]
EXPRESS WASH
express wash
indexOf("Wash") = 8
contains("press") = true
Express Iron
endsWith("dog") = true
number of words = 9
equalsIgnoreCase: true
```

Take them one at a time.

- **`startsWith(prefix)`** answers a yes/no question: does the string begin with these characters? It returns a `boolean`, which is why it can sit directly inside an `if`. Here the sentence starts with `the`, so we see `It does!`. `endsWith` asks the same about the end.
- **`substring(begin)`** and **`substring(begin, end)`** cut out part of a string. The first form takes everything from position `begin` to the end: `"HelloYarik".substring(5)` is `"Yarik"`. The second form takes the characters from `begin` up to *but not including* `end`: `substring(0, 5)` is positions 0, 1, 2, 3 and 4, which is `"Hello"`. The rule "include the start, exclude the end" looks odd at first, but it has a lovely property: the length of the piece is simply `end - begin`, and `substring(0, 5)` and `substring(5)` fit together exactly with no overlap and no gap.
- **`charAt(index)`** returns the single `char` at a position: `"abcde".charAt(3)` is `'d'`.
- **`length()`** returns the number of characters as an `int`.

- `trim()` removes spaces at both ends; `toUpperCase()` and `toLowerCase()` change the case; `indexOf(text)` finds where a piece of text first occurs (or returns `-1` if it does not occur); `contains(text)` asks whether it occurs at all; `replace(old, new)` swaps every occurrence of one piece for another; `split(" ")` cuts the string at every space and returns the pieces; `equalsIgnoreCase` compares two strings without caring about capitals.

Python writes most of these with operators instead of methods, as **Listing 3.16** shows.

Listing 3.16 `string_basics.py`: the same operations with Python's `str`.

```
def main():
    my_string = "the quick brown fox jumps over the lazy dog"
    if my_string.startswith("the"):
        print("It does!")
    else:
        print("It doesn't!")

    my_second_string = "HelloYarik"
    rest = my_second_string[5:]         # from index 5 to the end
    first_word = my_second_string[0:5]  # indices 0,1,2,3,4
    print(rest + " / " + first_word)

    another_string = "abcde"
    character_three = another_string[3]
    print("[3] = " + character_three)
    print("len() = " + str(len(another_string)))

    service = "  Express Wash  "
    print("[" + service.strip() + "]")
    print(service.strip().upper())
    print(service.strip().lower())
    print('find("Wash") = ' + str(service.strip().find("Wash")))
    print('"press" in service = ' + str("press" in service))
    print(service.strip().replace("Wash", "Iron"))
    print('endswith("dog") = ' + str(my_string.endswith("dog")))

    words = my_string.split(" ")
    print("number of words = " + str(len(words)))
    print("casefold compare: " + str("WASH".casefold() == "wash".casefold()))


if __name__ == "__main__":
    main()
```

```
It does!
Yarik / Hello
[3] = d
len() = 5
[Express Wash]
EXPRESS WASH
express wash
find("Wash") = 8
"press" in service = True
Express Iron
endswith("dog") = True
number of words = 9
casefold compare: True
```

Python uses square brackets for positions, `s[3]`, and for pieces, `s[5:]` and `s[0:5]`. These **slices** follow the same "include the start, exclude the end" rule as Java's `substring`. The length is a built-in function, `len(s)`, rather than a method. "Does it contain?" is the operator `in`. Python also allows *negative* positions, counted from the end: `s[-1]` is the last character, and `s[-3:]` is the last three. The Laundry class in Section 3.12 uses this to show only the last digits of a phone number.

Table 3.4 is a summary to keep at hand.

Table 3.4 Everyday string operations in Java and Python.

Task	Java (`String s`)	Python (`str s`)
Length	`s.length()`	`len(s)`
Character at position `i`	`s.charAt(i)`	`s[i]`
From `i` to the end	`s.substring(i)`	`s[i:]`
From `i` up to, not including, `j`	`s.substring(i, j)`	`s[i:j]`
Last character	`s.charAt(s.length() - 1)`	`s[-1]`
Starts / ends with	`s.startsWith(p)`, `s.endsWith(p)`	`s.startswith(p)`, `s.endswith(p)`
Contains	`s.contains(t)`	`t in s`
First position of `t`	`s.indexOf(t)` (`-1` if absent)	`s.find(t)` (`-1` if absent)
Upper / lower case	`s.toUpperCase()`, `s.toLowerCase()`	`s.upper()`, `s.lower()`
Remove outer spaces	`s.trim()` or `s.strip()`	`s.strip()`
Replace	`s.replace(a, b)`	`s.replace(a, b)`
Split into words	`s.split(" ")`	`s.split(" ")` or `s.split()`
Repeat	`s.repeat(3)`	`s * 3`
Same content	`s.equals(t)`	`s == t`

3.5.3 Strings never change

Here is a fact about strings that surprises almost everyone: **a string, once created, can never be changed**, in Java or in Python. Strings are **immutable**. Every method that seems to change a string, `toUpperCase`, `trim`, `replace`, actually builds and returns a *new* string and leaves the original exactly as it was. **Listing 3.17** shows it.

Listing 3.17 `StringImmutability.java`: string methods return new strings; `==` compares arrows while `equals` compares characters.

```
public class StringImmutability {

    public static void main(String[] args) {
        String s = "laundry";
        String t = s.toUpperCase();   // builds a NEW string; s is unchanged
        System.out.println("s = " + s + ", t = " + t);

        s = s + " day";               // s now points to another new string
        System.out.println("s = " + s);

        String typed = new String("wash");  // a separate object with the same text
        String code = "wash";
        System.out.println("code == typed:      " + (code == typed));
        System.out.println("code.equals(typed): " + code.equals(typed));

        StringBuilder receipt = new StringBuilder();
        for (int i = 1; i <= 3; i++) {
            receipt.append("item ").append(i).append("; ");
        }
        System.out.println(receipt.toString());
    }
}
```

```
s = laundry, t = LAUNDRY
s = laundry day
code == typed:      false
code.equals(typed): true
item 1; item 2; item 3;
```

After `String t = s.toUpperCase();`, the variable `s` still refers to `"laundry"` in lowercase: `toUpperCase` built a new string, `"LAUNDRY"`, and `t` refers to that. Then `s = s + " day";` looks as if it changes `s`, but it does not change any string. It builds a third string, `"laundry day"`, and moves the arrow `s` to point at it. The old `"laundry"` is still there, unchanged, with no name left pointing at it, and in time the garbage collector removes it. **Figure 3.12** draws the three strings and the arrows.

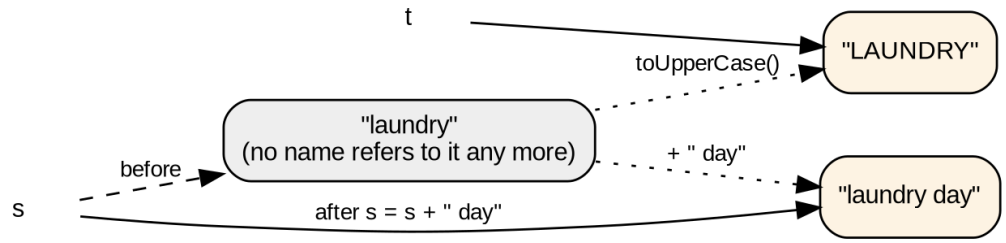


Figure 3.12 String methods never change a string: `toUpperCase` and `+` build new strings, and assignment only moves the arrow.

Why design strings this way? Because it makes them safe to share: when a `Customer` hands out its name through a getter, no caller can scramble the letters. Chapter 13 returns to immutability as a design tool for your own classes.

Immutability has one cost: building a long text by repeated `+` in a loop copies all the characters so far at every step. For many pieces, Java offers the mutable `StringBuilder`, and Python collects the pieces in a list and joins them once with `"".join(parts)`; the ends of **Listing 3.17** and **Listing 3.18** show both.

Now the most important line in **Listing 3.17**: `code == typed` is `false`, even though both hold `"wash"`. By now you can explain it with **Table 3.3**: `==` on references compares arrows, and `typed`

was built with `new String("wash")` as a separate object. **Figure 3.13** draws it. The method `equals` compares the characters one by one and says `true`.

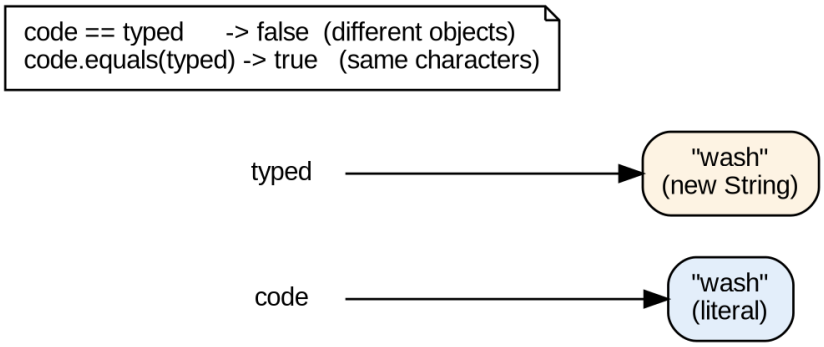


Figure 3.13 Two different `String` objects with the same characters: `==` says `false`, `equals` says `true`.

> **WARNING.** In Java, always compare the *content* of strings with `equals`, never with `==`. The trap is cruel because `==` sometimes appears to work: Java keeps one shared copy of every string literal written in the source code, so `"wash" == "wash"` happens to be `true`. But text that arrives at run time, typed by a user, read from a file, or built by `substring` or `+`, is a fresh object, and `==` will say `false` even when the characters match exactly.

Python avoids the trap, because Python's `==` compares content. **Listing 3.18** shows the same story; to make a string that is certainly a separate object, it builds `"wash"` from its letters at run time.

Listing 3.18 `string_immutability.py`: Python strings are immutable too; `==` compares content and `is` compares identity.

```
def main():
    s = "laundry"
    t = s.upper()  # builds a NEW string; s is unchanged
    print("s = " + s + ", t = " + t)

    s = s + " day"  # s now names another new string
    print("s = " + s)

    typed = "".join(["w", "a", "s", "h"])  # built at run time: a separate object
    code = "wash"
    print("code is typed: " + str(code is typed))
    print("code == typed: " + str(code == typed))

    parts = []
    for i in range(1, 4):
        parts.append("item " + str(i) + "; ")
    receipt = "".join(parts)
    print(receipt)


if __name__ == "__main__":
    main()
```

```
s = laundry, t = LAUNDRY
s = laundry day
code is typed: False
code == typed: True
item 1; item 2; item 3;
```

3.5.4 Formatting output

So far we have built output by gluing strings with `+`. That works, but it gives no control over how numbers look. A weight of `3.456` kg should appear on a receipt as `3.46`; a price of `24192` rupiah reads better as `24,192`; a column of prices should line up. For this both languages offer **format strings**: a template with placeholders, and a list of values to put into them. **Listing 3.19** and **Listing 3.20** show the most useful placeholders.

Listing 3.19 `StringFormatting.java`: `printf` and `String.format` with widths, decimals and thousands separators.

```java
public class StringFormatting {

    public static void main(String[] args) {
        String name = "Siti";
        double weightKg = 3.456;
        long priceRupiah = 24192;

        System.out.println("Customer " + name + " brought " + weightKg + " kg");
        System.out.printf("Customer %s brought %.2f kg%n", name, weightKg);
        System.out.printf("Total: Rp%,d%n", priceRupiah);
        System.out.printf("|%-10s|%8.1f|%n", "Wash", 7000.0);
        System.out.printf("|%-10s|%8.1f|%n", "Iron", 5000.0);

        String line = String.format("%s owes Rp%,d", name, priceRupiah);
        System.out.println(line + " (" + line.length() + " characters)");
        System.out.println("Percent sign: " + String.format("%.1f%%", 2.3));
    }
}
```

```
Customer Siti brought 3.456 kg
Customer Siti brought 3.46 kg
Total: Rp24,192
|Wash      |  7000.0|
|Iron      |  5000.0|
Siti owes Rp24,192 (18 characters)
Percent sign: 2.3%
```

Listing 3.20 `string_formatting.py`: the same formatting with Python's f-strings and `str.format`.

```python
def main():
    name = "Siti"
    weight_kg = 3.456
    price_rupiah = 24192

    print("Customer " + name + " brought " + str(weight_kg) + " kg")
    print(f"Customer {name} brought {weight_kg:.2f} kg")
    print(f"Total: Rp{price_rupiah:,d}")
    print(f"|{'Wash':<10}|{7000.0:8.1f}|")
    print(f"|{'Iron':<10}|{5000.0:8.1f}|")

    line = "{} owes Rp{:,d}".format(name, price_rupiah)
    print(line + " (" + str(len(line)) + " characters)")
    print("Percent sign: " + f"{2.3:.1f}%")


if __name__ == "__main__":
    main()
```

```
Customer Siti brought 3.456 kg
Customer Siti brought 3.46 kg
Total: Rp24,192
|Wash      |  7000.0|
|Iron      |  5000.0|
Siti owes Rp24,192 (18 characters)
Percent sign: 2.3%
```

In Java, `System.out.printf(template, values...)` prints and `String.format(template, values...)` returns the text instead. Each placeholder starts with `%`. In Python, an **f-string**, a string literal with the letter `f` in front, holds the values directly between braces, followed by a colon and a format specification: `f"{weight_kg:.2f}"`. **Table 3.5** lists the placeholders we use in this book.

Table 3.5 The format placeholders used in this book.

Meaning	Java placeholder	Python f-string	Example output
Text	`%s`	`{name}`	`Siti`
Whole number	`%d`	`{n:d}` or `{n}`	`24192`
With thousands separator	`%,d`	`{n:,d}`	`24,192`
Decimal, 2 places	`%.2f`	`{x:.2f}`	`3.46`
Right-aligned in 8 columns	`%8.1f`	`{x:8.1f}`	`7000.0`
Left-aligned in 10 columns	`%-10s`	`{s:<10}`	`Wash`
New line	`%n`	(`print` adds it)	
A literal percent sign	`%%`	`%`	`2.3%`

Two details deserve a warning. First, Java's `%,d` uses the thousands separator of your computer's **locale** (its regional settings): on a computer set to Indonesian it prints `24.192`, with a dot, as an Indonesian reader expects. The outputs in this book come from a computer with English settings, and Python's `{:,d}` always uses a comma. Second, in a Java format string the percent sign is special, so a literal `%` is written `%%`.

3.5.5 From text to numbers, and back

Everything a user types arrives as text. The word `"4"` on the command line is a string, not the number 4, and you cannot do arithmetic on it. Before calculating, you must **parse** it: read the characters and build the number they describe. Java provides `Integer.parseInt` for whole numbers and `Double.parseDouble` for decimals; Python uses `int(...)` and `float(...)`. Going the other way, `String.valueOf(n)` in Java and `str(n)` in Python turn a number into text. **Listing 3.21** and **Listing 3.22** show both directions.

Listing 3.21 `ParseNumbers.java`: turning command-line text into an `int` and a `double`.

```java
public class ParseNumbers {

    public static void main(String[] args) {
        String kgText = args.length > 0 ? args[0] : "4";
        String rateText = args.length > 1 ? args[1] : "7500.5";

        int kg = Integer.parseInt(kgText);           // text -> int
        double rate = Double.parseDouble(rateText);  // text -> double
        System.out.println(kg + " kg at " + rate + " per kg = " + (kg * rate));

        String back = String.valueOf(kg) + "!";      // int -> text
        System.out.println(back);
        System.out.println("Text \"12\" + \"3\" = " + ("12" + "3"));
        System.out.println("Number 12 + 3 = " + (Integer.parseInt("12") + 3));
    }
}
```

```
4 kg at 7500.5 per kg = 30002.0
4!
Text "12" + "3" = 123
Number 12 + 3 = 15
```

The first two lines of `main` use a compact form of `if` called the **conditional operator**: `condition ? a : b` is worth `a` when the condition is true and `b` otherwise. So `args.length > 0 ? args[0] : "4"` means "the first command-line word if there is one, else the text `"4"`". Python writes the same idea in words, `a if condition else b`, as **Listing 3.22** shows. `Customer` in Section 3.12 uses it once more to choose between the words *member* and *guest*.

With two words on the command line, `java ParseNumbers 6 7000`, the program uses them instead of its defaults:

```
6 kg at 7000.0 per kg = 42000.0
6!
Text "12" + "3" = 123
Number 12 + 3 = 15
```

Listing 3.22 `parse_numbers.py`: `int(...)` and `float(...)` parse text in Python.

```python
import sys


def main():
    kg_text = sys.argv[1] if len(sys.argv) > 1 else "4"
    rate_text = sys.argv[2] if len(sys.argv) > 2 else "7500.5"

    kg = int(kg_text)          # text -> int
    rate = float(rate_text)    # text -> float
    print(str(kg) + " kg at " + str(rate) + " per kg = " + str(kg * rate))

    back = str(kg) + "!"       # int -> text
    print(back)
    print('Text "12" + "3" = ' + ("12" + "3"))
    print("Number 12 + 3 = " + str(int("12") + 3))


if __name__ == "__main__":
    main()
```

```
4 kg at 7500.5 per kg = 30002.0
4!
Text "12" + "3" = 123
Number 12 + 3 = 15
```

The last two lines of each output make the point sharply. `"12" + "3"` joins two strings and gives `"123"`; `Integer.parseInt("12") + 3` adds two numbers and gives `15`. Same symbol, different meaning, decided by the types on either side.

What if the text is not a number? Parsing fails loudly. Running `java ParseNumbers 3 abc` stops with

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
    at
java.base/jdk.internal.math.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2054)
    at java.base/jdk.internal.math.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
    at java.base/java.lang.Double.parseDouble(Double.java:792)
    at ParseNumbers.main(ParseNumbers.java:8)
```

and the Python version ends its error report with

```
ValueError: could not convert string to float: 'abc'
```

Read the Java message from the bottom up: the error happened in `main`, at line 8 of `ParseNumbers.java`, inside a call to `Double.parseDouble`, because the input string was `"abc"`. Chapter 5 shows how to catch such errors and ask the user again instead of stopping.

3.6 Types

3.6.1 Java's eight primitive types

Every value in a Java program has a **type**, and the type decides two things: which values are allowed, and which operations make sense. You cannot multiply two dogs, and you cannot call `toUpperCase` on the number 7.

We have met the reference types, of which there are as many as there are classes. The primitive types are a small, fixed family of eight, listed in **Table 3.6** together with the memory each needs and the range of values it can hold [6].

Table 3.6 Java's eight primitive types, and the Python type that plays each role.

Type	Holds	Size	Range	Python counterpart
`byte`	whole number	8 bits	−128 to 127	`int`
`short`	whole number	16 bits	−32 768 to 32 767	`int`
`int`	whole number	32 bits	−2 147 483 648 to 2 147 483 647	`int`
`long`	whole number	64 bits	about $\pm 9.22 \times 10^{18}$	`int`
`float`	decimal number	32 bits	about 7 significant digits	(none)
`double`	decimal number	64 bits	about 15 to 16 significant digits	`float`
`char`	one character	16 bits	`'\u0000'` to `'\uffff'`	`str` of length 1
`boolean`	truth value	(unspecified)	`true` or `false`	`bool`

There are two kinds of numbers: **whole numbers** (integers) and **decimals**.

For integers there are four types, `byte`, `short`, `int` and `long`, which differ only in how many bits they use and therefore how large a number they can hold. A type with b bits can represent 2^b different patterns; Java spends half of them on negative numbers and half on zero and the positive numbers. So the range of a b-bit integer type is

$$-2^{b-1} \le v \le 2^{b-1} - 1 \tag{3.1}$$

which, by Equation (3.1), gives $-2^7 = -128$ to $2^7 - 1 = 127$ for the 8-bit `byte` and -2^{31} to $2^{31} - 1$, a little over two billion either way, for the 32-bit `int`. The smaller types save memory, which matters when you store millions of numbers on a small device, but most of the time the difference is not noticeable. Most Java methods return `int` or `long`, so those are the two you will use. The program in **Listing 3.27** prints the ranges straight from Java's own constants.

For decimals there are `float`, 32 bits with about 7 significant digits, and `double`, 64 bits with about twice as many (hence "double precision"). Most Java methods, such as `Math.sqrt` and `Math.pow`, return `double`, so unless you have a compelling reason, choose `double`.

The remaining two types are `char`, a single character, and `boolean`, which stores `true` or `false` and nothing else. `boolean` is what every condition in an `if` or a loop produces.

Python is much simpler. It has one integer type, `int`, with **no upper limit**: it grows to as many digits as needed. It has one decimal type, `float`, which is exactly Java's `double`. It has `bool` for truth values, and it has no character type at all.

3.6.2 Static typing and dynamic typing

The biggest difference between Java and Python is not a keyword or a bracket. It is *when* types are checked.

Java is **statically typed**. Every variable is declared with a type, and it may hold only values of that type, for its whole life. The compiler checks every assignment and every method call *before the program runs*. **Listing 3.23** shows the rules at work.

Listing 3.23 `StaticTyping.java`: in Java every variable has one type for life; `var` lets the compiler work the type out.

```
public class StaticTyping {

    public static void main(String[] args) {
        int count = 3;              // count may only ever hold int values
        double price = 7000;        // the int 7000 is widened to 7000.0
        var service = "Wash";       // var: the compiler infers String
        long total = count * 7000L;

        System.out.println(count + " " + price + " " + service + " " + total);
        System.out.println(service.getClass().getSimpleName());
        // count = "three";  // would not compile: String cannot become int
    }
}
```

```
3 7000.0 Wash 21000
String
```

A few details in **Listing 3.23** are worth noticing. `double price = 7000;` is allowed: an `int` can be **widened** to a `double` with no loss. The reverse, `int n = 7000.5;`, is refused, because it would lose the fraction. `var service = "Wash";` does not make `service` typeless: `var` simply asks the compiler to *infer* the type from the right-hand side, and `service` is a `String` for ever after. And `7000L`, with an `L`, is a `long` literal, which makes `count * 7000L` a `long` multiplication.

Now remove the `//` in front of the line `count = "three";` and compile again. The compiler refuses, and the program never runs:

```
StaticTyping.java:11: error: incompatible types: String cannot be converted to int
        count = "three";
                ^
1 error
```

Python is **dynamically typed**. Variables have no declared type; a name can be bound to a value of any type, and it can be rebound to a value of a different type later. The *values* still have types (every object knows whether it is an `int` or a `str`), and they are checked, but only at the moment an operation runs. **Listing 3.24** shows the consequences.

Listing 3.24 `dynamic_typing.py`: in Python a name can refer to values of different types; types are checked when an operation runs.

```
def main():
    count = 3
    print(count, type(count).__name__)
    count = "three"          # allowed: the NAME count now refers to a str
    print(count, type(count).__name__)

    price: float = 7000      # a type hint documents intent; Python does not enforce it
    print(price, type(price).__name__)
    total = 3 * 7000
    print(total, type(total).__name__, isinstance(total, int))

    try:
        print("Total: " + total)
    except TypeError as error:
        print("TypeError:", error)


if __name__ == "__main__":
    main()
```

```
3 int
three str
7000 int
21000 int True
TypeError: can only concatenate str (not "int") to str
```

The name `count` first refers to an `int` and then to a `str`, and Python does not object. The annotation `price: float = 7000` is a **type hint**: editors and type checkers read it, but Python itself does not enforce it, which is why `type(price)` is still `int`. Finally, `"Total: " + total` fails with a `TypeError` *at run time*: Python will not guess whether you meant to add numbers or join text, which is why `str(...)` was needed in Section 3.3.5. Python is *dynamically* typed but *strongly* typed: types are checked late, but they are checked.

Neither style is simply better: static typing catches mistakes before the program runs and helps in large programs; dynamic typing makes small programs shorter to write.

3.6.3 Characters and truth values

Listing 3.25 looks at the two remaining primitive types.

Listing 3.25 `CharBoolean.java`: a `char` is a small number underneath; `&&`, `||` and `!` combine truth values.

```java
public class CharBoolean {

    public static void main(String[] args) {
        char grade = 'B';
        char next = (char) (grade + 1);      // characters are small numbers underneath
        System.out.println(grade + " is code " + (int) grade + "; next is " + next);

        boolean isMember = true;
        boolean isExpress = false;
        double weightKg = 2.5;
        boolean discount = isMember && weightKg >= 2.0;
        boolean surcharge = isExpress || weightKg > 10.0;
        System.out.println("discount = " + discount + ", surcharge = " + surcharge);
        System.out.println("!isMember = " + !isMember);

        String note = null;
        // && stops early: note.length() is never called when note is null
        if (note != null && note.length() > 0) {
            System.out.println("Note: " + note);
        } else {
            System.out.println("No note on this order");
        }
    }
}
```

```
B is code 66; next is C
discount = true, surcharge = false
!isMember = false
No note on this order
```

A `char` is stored as a number, its **character code** in the Unicode table: `'B'` is 66. Adding 1 gives 67, and casting back with `(char)` gives `'C'`. This is how programs step through the alphabet.

Truth values combine with three **logical operators**. `a && b` (**and**) is `true` only when both are true. `a || b` (**or**) is `true` when at least one is true. `!a` (**not**) turns `true` into `false` and back. In **Listing 3.25** a member with 2.5 kg of laundry qualifies for a discount (`true && true`), and the order does not need a surcharge (`false || false`).

The operators `&&` and `||` are **short-circuit** operators: they stop as soon as the answer is known. If the left side of `&&` is `false`, the whole expression must be `false`, so the right side is never evaluated. The test `note != null && note.length() > 0` relies on this: when `note` is `null`, Java never calls `note.length()`, and so never meets a `NullPointerException`.

Listing 3.26 `char_boolean.py`: `ord` and `chr` convert between characters and codes; `and`, `or` and `not` combine truth values.

```
def main():
    grade = "B"  # Python has no char type: a one-character string
    next_grade = chr(ord(grade) + 1)
    print(grade + " is code " + str(ord(grade)) + "; next is " + next_grade)

    is_member = True
    is_express = False
    weight_kg = 2.5
    discount = is_member and weight_kg >= 2.0
    surcharge = is_express or weight_kg > 10.0
    print("discount = " + str(discount) + ", surcharge = " + str(surcharge))
    print("not is_member = " + str(not is_member))

    note = None
    # "and" stops early: len(note) is never called when note is None
    if note is not None and len(note) > 0:
        print("Note: " + note)
    else:
        print("No note on this order")

    print("bool(0) = " + str(bool(0)) + ", bool('') = " + str(bool(""))
          + ", bool('x') = " + str(bool("x")))


if __name__ == "__main__":
    main()
```

```
B is code 66; next is C
discount = True, surcharge = False
not is_member = False
No note on this order
bool(0) = False, bool('') = False, bool('x') = True
```

Listing 3.26 shows that Python writes the operators as words, `and`, `or`, `not`, and they short-circuit in the same way. Python's `ord` gives the code of a one-character string and `chr` goes back. One Python feature has no Java counterpart: **truthiness**. In Python any value can stand in a condition: `0`, the empty string `""`, `None` and empty collections count as false, everything else as true. Java allows only real `boolean` values in a condition, and `if (count)` does not compile.

3.7 Arithmetic and Its Pitfalls

3.7.1 Operators and precedence

Arithmetic in a program works the way it works on a calculator, with a few additions. Suppose

```
int i = 0;
int j = 3;
int result = 2 + i * j;
```

Java multiplies `i` by `j` first and then adds 2, exactly as mathematics does, so `result` is 2. Multiplication and division bind more tightly than addition and subtraction; this ordering is called **precedence**. Parentheses override it: `(2 + i) * j` is 6. **Table 3.7** lists the arithmetic operators, from tightest to loosest.

Table 3.7 Common operators, from highest precedence to lowest. In Python, `**` binds more tightly than unary minus, so `-2 ** 2` is `-4`.

Operator	Meaning	Java example	Python example
`( )`	grouping	`(2 + i) * j`	`(2 + i) * j`
`-x`, `!x` / `not x`	negation	`-x`, `!ok`	`-x`, `not ok`
`*`, `/`, `%`	multiply, divide, remainder	`7 / 2` is `3`	`7 / 2` is `3.5`, `7 // 2` is `3`
`**`	power	(use `Math.pow`)	`2 ** 3` is `8`
`+`, `-`	add, subtract (and join strings)	`"a" + 1` is `"a1"`	`"a" + str(1)`
`<`, `<=`, `>`, `>=`	comparison	`i > 0`	`i > 0`
`==`, `!=`	equal, not equal	`i == 0`	`i == 0`
`&&` / `and`	logical and	`a && b`	`a and b`
`||` / `or`	logical or	`a || b`	`a or b`
`=`, `+=`, `-=`, …	assignment	`x += 3`	`x += 3`

Java has no power operator. For 2^3 you call `Math.pow(2, 3)`, which takes the base and the exponent and returns the `double` `8.0`. Python writes `2 ** 3` and, because both operands are integers, gives the integer 8.

Java has two tiny operators that you will see in every loop: `i++` adds one to `i`, and `i--` subtracts one. Python does not have them; it writes `i += 1` and `i -= 1`. Both languages have the compound assignments `+=`, `-=`, `*=`, `/=`: `x += 3` means `x = x + 3`.

Computer arithmetic *differs* from school arithmetic in three places, and each has cost real money in real software.

3.7.2 Pitfall one: integer overflow

The limits in **Table 3.6** apply at *every step* of a calculation, not only to the answer at the end. Consider:

```
int x = 100000;   // 100,000
int y = 200000;   // 200,000
int z = 5000;     // 5,000
System.out.println(x * y / z);
```

By hand: $100\,000 \times 200\,000 = 20\,000\,000\,000$, and divided by $5\,000$ that is $4\,000\,000$. None of the three variables is anywhere near the limit of an `int`, and neither is the answer. Yet **Listing 3.27** prints something else entirely.

Listing 3.27 `IntegerTypes.java`: the ranges of the integer types, and an overflow in the middle of a calculation.

```java
public class IntegerTypes {

    public static void main(String[] args) {
        System.out.println("byte  : " + Byte.MIN_VALUE + " .. " + Byte.MAX_VALUE);
        System.out.println("short : " + Short.MIN_VALUE + " .. " + Short.MAX_VALUE);
        System.out.println("int   : " + Integer.MIN_VALUE + " .. " + Integer.MAX_VALUE);
        System.out.println("long  : " + Long.MIN_VALUE + " .. " + Long.MAX_VALUE);

        int x = 100000;   // 100,000
        int y = 200000;   // 200,000
        int z = 5000;     // 5,000
        System.out.println("x * y / z as int : " + (x * y / z));
        System.out.println("x * y as int     : " + (x * y));
        System.out.println("x * y / z as long: " + ((long) x * y / z));

        int biggest = Integer.MAX_VALUE;
        System.out.println("MAX_VALUE + 1    : " + (biggest + 1));
    }
}
```

```
byte  : -128 .. 127
short : -32768 .. 32767
int   : -2147483648 .. 2147483647
long  : -9223372036854775808 .. 9223372036854775807
x * y / z as int : -294967
x * y as int     : -1474836480
x * y / z as long: 4000000
MAX_VALUE + 1    : -2147483648
```

The answer is $-294\,967$: negative, and nonsense. The culprit is the intermediate product $x \cdot y = 2 \times 10^{10}$, which is far above the `int` limit of $2^{31} - 1 = 2\,147\,483\,647$. Java does not stop or warn. It keeps only the lowest 32 bits of the true result, which amounts to taking the true value modulo 2^{32} and reading the result as a signed number:

$$\text{stored}(v) = ((v + 2^{31}) \bmod 2^{32}) - 2^{31} \tag{3.2}$$

Let us verify Equation (3.2) by hand for $v = 2 \times 10^{10}$. First, $2^{32} = 4\,294\,967\,296$. Then $v + 2^{31} = 20\,000\,000\,000 + 2\,147\,483\,648 = 22\,147\,483\,648$. Dividing by 2^{32} goes 5 whole times, because $5 \times 4\,294\,967\,296 = 21\,474\,836\,480$, with remainder $22\,147\,483\,648 - 21\,474\,836\,480 = 672\,647\,168$. Finally $672\,647\,168 - 2\,147\,483\,648 = -1\,474\,836\,480$. That is exactly the "x * y as int" line of the output. Dividing by 5 000, and dropping the fraction as integer division does (Section 3.7.4), gives $-294\,967$. The mystery is solved: the product **overflowed**.

The same wrap-around explains the last line: `Integer.MAX_VALUE + 1` does not give $2\,147\,483\,648$ but jumps to the most negative `int`, $-2\,147\,483\,648$, like a car odometer rolling over from 999999 to 000000.

The cure is to do the risky arithmetic in a wider type. Writing `(long) x * y / z` converts `x` to a `long` *before* the multiplication, so the product is computed in 64 bits, where 2×10^{10} fits easily, and the answer is the correct 4 000 000. Note the placement: `(long) (x * y)` would be too late, because the damage is done inside the parentheses, in `int` arithmetic, before the conversion.

Python sidesteps the problem because its `int` has no limit, as **Listing 3.28** shows. To see what Java would have computed, the listing imitates a 32-bit `int` with a small function that applies Equation (3.2) using bit operations.

Listing 3.28 `integer_types.py`: Python integers never overflow; `wrap_int32` imitates Java's 32-bit `int`.

```
def wrap_int32(value):
    """Squeeze a Python int into 32 bits the way Java's int does."""
    value = value & 0xFFFFFFFF          # keep only the lowest 32 bits
    if value >= 2**31:                  # the top bit means "negative"
        value -= 2**32
    return value


def main():
    x = 100000   # 100,000
    y = 200000   # 200,000
    z = 5000     # 5,000
    print("x * y // z          : " + str(x * y // z))
    print("x * y               : " + str(x * y))
    print("x * y wrapped to 32 : " + str(wrap_int32(x * y)))
    print("wrapped, then // z  : " + str(int(wrap_int32(x * y) / z)))

    biggest = 2**31 - 1
    print("2**31 - 1 + 1       : " + str(biggest + 1))
    print("wrapped             : " + str(wrap_int32(biggest + 1)))
    print("2**100              : " + str(2**100))


if __name__ == "__main__":
    main()
```

```
x * y // z          : 4000000
x * y               : 20000000000
x * y wrapped to 32 : -1474836480
wrapped, then // z  : -294967
2**31 - 1 + 1       : 2147483648
wrapped             : -2147483648
2**100              : 1267650600228229401496703205376
```

> **WARNING.** Overflow is silent. Whenever a calculation multiplies large whole numbers, such as money in rupiah (where a price of a few million times a quantity of a few thousand already exceeds two billion), do it in `long`. In Java, `Math.multiplyExact(a, b)` is a safer alternative that stops the program with an error instead of returning a wrong answer.

3.7.3 Pitfall two: the computer cannot count to one in tenths

Computers work with whole numbers of bits, so **floating-point** numbers store a decimal as a whole-number *significand* and a power of two, much as scientific notation stores 6.02×10^{23} [7]. A number as innocent as 0.1 has no exact representation in base two, just as $\frac{1}{3}$ has none in base ten (0.3333 … must stop somewhere). The computer stores the nearest representable number, and tiny errors creep in [8].

Listing 3.29 tries adding 0.1 ten times. The answer should be 1.

Listing 3.29 `FloatingPoint.java`: adding 0.1 ten times, and why decimal numbers must not be compared with `==`.

```
public class FloatingPoint {

    public static void main(String[] args) {
        System.out.println(0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f +
0.1f); // floats
        System.out.println(0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1);
// doubles

        System.out.println("0.1 + 0.2 + 0.3 = " + (0.1 + 0.2 + 0.3));
        System.out.println("0.2 + 0.3 + 0.1 = " + (0.2 + 0.3 + 0.1));

        double a = 0.1 + 0.2;
        double b = 0.3;
        System.out.println("a == b            : " + (a == b));
        System.out.println("|a - b| < 1e-9    : " + (Math.abs(a - b) < 1e-9));
        System.out.println("2^3 with Math.pow : " + Math.pow(2, 3));
    }
}
```

```
1.0000001
0.9999999999999999
0.1 + 0.2 + 0.3 = 0.6000000000000001
0.2 + 0.3 + 0.1 = 0.6
a == b            : false
|a - b| < 1e-9    : true
2^3 with Math.pow : 8.0
```

Neither line prints `1.0`. The `float` sum overshoots to `1.0000001`; the `double` sum falls just short at `0.9999999999999999`. The `double` error is far smaller, because a `double` has more bits, but it is not zero. **Figure 3.14** follows the error as more tenths are added: it grows, it wobbles, and for `float` it is roughly a billion times larger than for `double`.

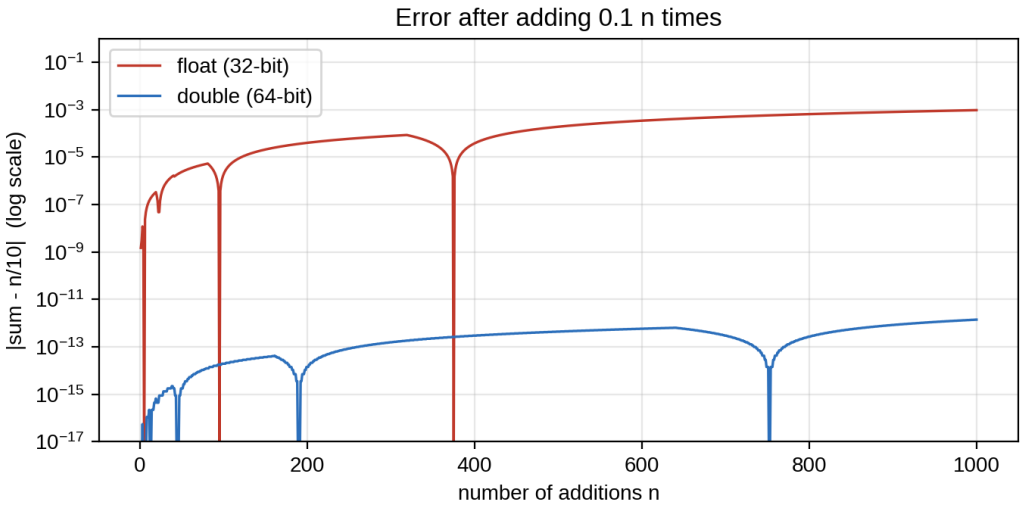


Figure 3.14 The error after adding 0.1 again and again, for `float` and `double`; the dips to the bottom are the moments when the rounding errors happen to cancel.

Two lessons follow. The first is that the usual laws of arithmetic are only approximately true. Addition is no longer exactly associative: `0.1 + 0.2 + 0.3` gives `0.6000000000000001`, while `0.2 + 0.3 + 0.1` gives `0.6`, because the rounding happens at different steps.

The second lesson is the rule that matters most in practice: **never compare two floating-point numbers with `==`**. Even when mathematics says they are equal, the computed values may differ in the last bit, and `0.1 + 0.2 == 0.3` is `false`. Compare them with a tolerance instead: two decimals a and b count as equal when

$$|a - b| < \varepsilon \tag{3.3}$$

for a small ε (the Greek letter epsilon) chosen to suit the size of the numbers involved, such as 10^{-9}. In Java that is `Math.abs(a - b) < 1e-9`, where `1e-9` is how programs write 10^{-9}. Equation (3.3) is exactly what `DogCheck` used for the weight in Section 3.3.6.

Listing 3.30 `floating_point.py`: the same effects in Python, with `math.isclose` and the `decimal` module.

```
import math
from decimal import Decimal


def main():
    print(0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1)

    print("0.1 + 0.2 + 0.3 = " + str(0.1 + 0.2 + 0.3))
    print("0.2 + 0.3 + 0.1 = " + str(0.2 + 0.3 + 0.1))

    a = 0.1 + 0.2
    b = 0.3
    print("a == b             : " + str(a == b))
    print("abs(a - b) < 1e-9  : " + str(abs(a - b) < 1e-9))
    print("math.isclose(a, b) : " + str(math.isclose(a, b)))
    print("2 ** 3, math.pow   : " + str(2 ** 3) + ", " + str(math.pow(2, 3)))
    print("Decimal sum        : " + str(Decimal("0.1") * 10))


if __name__ == "__main__":
    main()
```

```
0.9999999999999999
0.1 + 0.2 + 0.3 = 0.6000000000000001
0.2 + 0.3 + 0.1 = 0.6
a == b             : False
abs(a - b) < 1e-9  : True
math.isclose(a, b) : True
2 ** 3, math.pow   : 8, 8.0
Decimal sum        : 1.0
```

Listing 3.30 shows that Python behaves exactly like Java's `double`, since its `float` *is* a 64-bit double. Python also offers `math.isclose(a, b)`, a ready-made version of Equation (3.3), and the `decimal` module, which computes in base ten and so represents 0.1 exactly: `Decimal("0.1") * 10` is exactly `1.0`.

> **KEY INSIGHT.** Money should never be stored as `double`. A price list whose totals are off by a fraction of a rupiah will, sooner or later, fail to balance. Store money as a whole number of the smallest unit, rupiah in a `long` (Java) or an `int` (Python), as the Profitina Laundry classes in Section 3.12 do, or use `BigDecimal` in Java and `Decimal` in Python.

3.7.4 Pitfall three: integer division

Sooner or later every beginner writes `1 / 16` and gets `0`. When both operands of `/` are integers, Java performs **integer division**: it divides and throws away the fraction. So `1 / 16` is `0`, not `0.0625`; `17 / 5` is `3`; `5 / 6` is `0`.

Its partner is the **remainder** operator `%`: `17 % 5` is 2, because $17 = 3 \times 5 + 2$. For any integers a and $b \neq 0$, the two operators are tied together by

$$a = (a/b) \cdot b + (a \bmod b) \tag{3.4}$$

where a/b means the integer quotient. Equation (3.4) is the rule you learned in primary school as "quotient and remainder". The remainder is far more useful than it first appears: `n % 2 == 0` tests whether `n` is even, `n % 10` is the last decimal digit of `n`, and `year % 100 == 0` picks out every hundredth year. Section 3.10 builds a whole algorithm out of `% 10` and `/ 10`.

To get the fractional answer, at least one operand must be a decimal. With constants, simply write one of them as a decimal: `1 / 16.0` is `0.0625`. With variables you cannot add `.0`, so you **cast**: write the target type in parentheses in front of the value, and Java treats the value as that type for the moment. `(double) x / y` converts `x` to a `double`, and then the division is a decimal division. As with overflow, the placement matters: `(double) (x / y)` divides first, in integers, and converts the already-truncated result, giving `0.0`. Casting a decimal to an integer, `(int) 3.99`, simply cuts off the fraction and gives `3`; to round to the nearest whole number, use `Math.round`. **Listing 3.31** and **Listing 3.32** try all of this.

Listing 3.31 `IntegerDivision.java`: integer division, the remainder operator, and casting.

```
public class IntegerDivision {

    public static void main(String[] args) {
        System.out.println("1 / 16    = " + (1 / 16));
        System.out.println("1 / 16.0  = " + (1 / 16.0));

        int x = 5;
        int y = 6;
        System.out.println("x / y          = " + (x / y));
        System.out.println("(double) x / y = " + ((double) x / y));
        System.out.println("(double) (x / y) = " + ((double) (x / y)));

        System.out.println("17 / 5 = " + (17 / 5) + ", 17 % 5 = " + (17 % 5));
        System.out.println("-7 / 2 = " + (-7 / 2) + ", -7 % 2 = " + (-7 % 2));
        System.out.println("(int) 3.99 = " + (int) 3.99);
        System.out.println("Math.round(3.5) = " + Math.round(3.5));
        int result = 2 + 0 * 3;          // multiplication before addition
        System.out.println("2 + 0 * 3 = " + result + ", (2 + 0) * 3 = " + ((2 + 0) * 3));
    }
}
```

```
1 / 16    = 0
1 / 16.0  = 0.0625
x / y          = 0
(double) x / y = 0.8333333333333334
(double) (x / y) = 0.0
17 / 5 = 3, 17 % 5 = 2
-7 / 2 = -3, -7 % 2 = -1
(int) 3.99 = 3
Math.round(3.5) = 4
2 + 0 * 3 = 2, (2 + 0) * 3 = 6
```

Listing 3.32 `integer_division.py`: Python separates true division `/` from floor division `//`.

```
def main():
    print("1 // 16  = " + str(1 // 16))
    print("1 / 16   = " + str(1 / 16))

    x = 5
    y = 6
    print("x // y   = " + str(x // y))
    print("x / y    = " + str(x / y))
    print("float(x // y) = " + str(float(x // y)))

    print("17 // 5 = " + str(17 // 5) + ", 17 % 5 = " + str(17 % 5))
    print("-7 // 2 = " + str(-7 // 2) + ", -7 % 2 = " + str(-7 % 2))
    print("int(-7 / 2) = " + str(int(-7 / 2)))
    print("int(3.99) = " + str(int(3.99)))
    print("round(3.5) = " + str(round(3.5)) + ", round(2.5) = " + str(round(2.5)))
    result = 2 + 0 * 3  # multiplication before addition
    print("2 + 0 * 3 = " + str(result) + ", (2 + 0) * 3 = " + str((2 + 0) * 3))


if __name__ == "__main__":
    main()
```

```
1 // 16  = 0
1 / 16   = 0.0625
x // y   = 0
x / y    = 0.8333333333333334
float(x // y) = 0.0
17 // 5 = 3, 17 % 5 = 2
-7 // 2 = -4, -7 % 2 = 1
int(-7 / 2) = -3
int(3.99) = 3
round(3.5) = 4, round(2.5) = 2
2 + 0 * 3 = 2, (2 + 0) * 3 = 6
```

Python took a cleaner route: its `/` *always* gives a decimal, `1 / 16` is `0.0625`, and a separate operator, `//`, performs integer (floor) division. That removes the trap but adds a difference to remember when you translate: **with negative numbers, Java and Python disagree**. Java truncates *toward zero*, so `-7 / 2` is `-3` and `-7 % 2` is `-1`; Python rounds *down*, so `-7 // 2` is `-4` and `-7 % 2` is `1`. Both obey Equation (3.4): $(-3) \cdot 2 + (-1) = -7$ and $(-4) \cdot 2 + 1 = -7$. A second surprise waits in Python's `round`, which rounds an exact half to the *even* neighbor: `round(2.5)` is `2` and `round(3.5)` is `4`, while Java's `Math.round` always rounds halves up.

Table 3.8 Division and rounding in Java and Python, side by side.

Expression	Java	Python
`1 / 16`	`0`	`0.0625`
`1 // 16`	(not valid)	`0`
`17 / 5` and `17 % 5`	`3` and `2`	`3.4` and `2`
`-7 / 2` (integer quotient)	`-3`	`-7 // 2` is `-4`
`-7 % 2`	`-1`	`1`
`(double) 5 / 6` / `5 / 6`	`0.8333333333333334`	`0.8333333333333334`
round 2.5	`Math.round(2.5)` is `3`	`round(2.5)` is `2`

Table 3.8 gathers the differences in one place.

3.8 Conditionals

3.8.1 Doing something only when a condition holds

Often a program should do something only when a certain condition is true: "*if* the customer is a member, *then* take 10% off, *otherwise* charge the full price". The statement for this is `if`. The general shape is

```
if (condition) {
    statements run when the condition is true
} else {
    statements run when it is false
}
```

The condition must be a `boolean` expression: a comparison like `i > 0`, a `boolean` variable, a method that returns `boolean` like `s.startsWith("the")`, or a combination of these with `&&`, `||` and `!`. The `else` part is optional. The braces group several statements into one **block**; Java allows you to omit them when a branch has a single statement, but always writing them is a good habit, because a second statement added later without braces silently falls outside the `if`.

Listing 3.33 starts with a small puzzle. Before looking at the output, decide what it prints, and then decide what it would print if `i` were 1.

Listing 3.33 `IfDemo.java`: an `if`/`else`, and an `else if` chain tried for several values.

```
public class IfDemo {

    public static void main(String[] args) {
        int i = 0;
        int j = 3;
        if (i > 0) {
            System.out.println("i is greater than 0!");
            System.out.println("j is " + (j - 1));
        } else {
            System.out.println("i is less than or equal to 0");
            System.out.println("j is " + j);
        }

        for (int k = 2; k <= 6; k++) {
            if (k > 5) {
                System.out.println(k + ": more than 5");
            } else if (k > 4) {
                System.out.println(k + ": more than 4");
            } else if (k > 3) {
                System.out.println(k + ": more than 3");
            } else {
                System.out.println(k + ": 3 or less");
            }
        }
    }
}
```

```
i is less than or equal to 0
j is 3
2: 3 or less
3: 3 or less
4: more than 3
5: more than 4
6: more than 5
```

With `i = 0`, the condition `i > 0` is false, so only the `else` block runs: it prints `i is less than or equal to 0` and then `j is 3`. If `i` were 1, the condition would be true, the first block would run,

and the program would print `i is greater than 0!` and then `j is 2`, because that block prints `j - 1`. Notice that exactly one of the two blocks runs, never both and never neither.

The Python version in **Listing 3.34** has the same logic with less punctuation: no parentheses around the condition, a colon at the end of the `if` line, and indentation to mark the block. Python writes "else if" as one word, `elif`.

Listing 3.34 `if_demo.py`: `if`, `elif` and `else` in Python.

```
def main():
    i = 0
    j = 3
    if i > 0:
        print("i is greater than 0!")
        print("j is " + str(j - 1))
    else:
        print("i is less than or equal to 0")
        print("j is " + str(j))

    for k in range(2, 7):
        if k > 5:
            print(str(k) + ": more than 5")
        elif k > 4:
            print(str(k) + ": more than 4")
        elif k > 3:
            print(str(k) + ": more than 3")
        else:
            print(str(k) + ": 3 or less")


if __name__ == "__main__":
    main()
```

```
i is less than or equal to 0
j is 3
2: 3 or less
3: 3 or less
4: more than 3
5: more than 4
6: more than 5
```

3.8.2 Chains of decisions

You can keep adding `else if` branches as long as you need. The second half of **Listing 3.33** tries such a chain for `k` from 2 to 6:

```
if (k > 5) { ... } else if (k > 4) { ... } else if (k > 3) { ... } else { ... }
```

The conditions are tested *in order, from the top*, and the first one that is true wins; the rest are skipped. That order matters. When `k` is 6, all three conditions are true, but only the first branch runs, printing `more than 5`. When `k` is 2 or 3, none is true, and the closing `else` catches it. **Figure 3.15** draws the chain as a flowchart, and **Table 3.9** records which test decides each value.

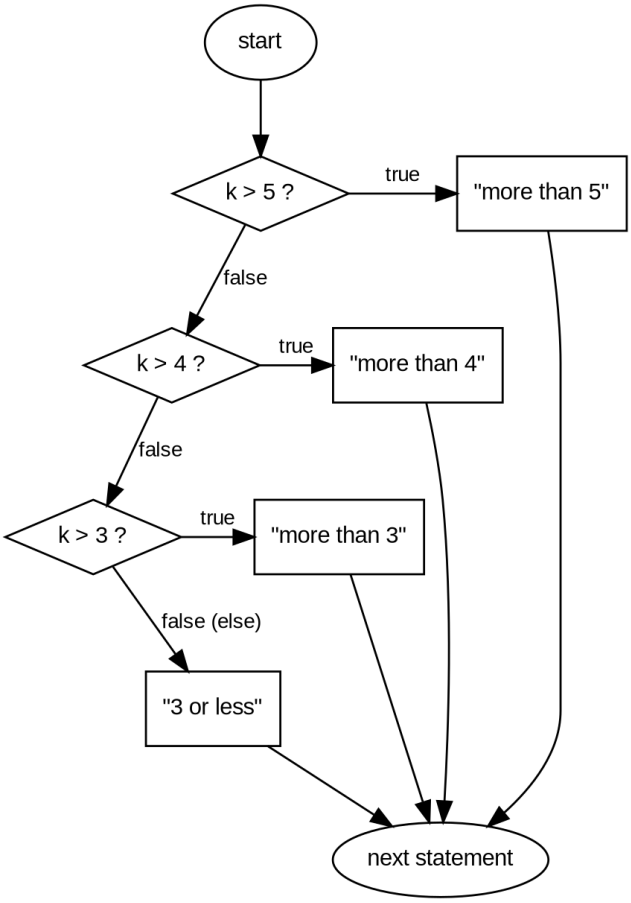


Figure 3.15 An `else if` chain as a flowchart: the conditions are tried from the top, and the first true one decides.

Table 3.9 Tracing the `else if` chain of **Listing 3.33** for $k = 2, \ldots, 6$.

k	k > 5?	k > 4?	k > 3?	Printed
2	false	false	false	`2: 3 or less`
3	false	false	false	`3: 3 or less`
4	false	false	true	`4: more than 3`
5	false	true	(not tested)	`5: more than 4`
6	true	(not tested)	(not tested)	`6: more than 5`

Had we written the tests in the opposite order, starting with `k > 3`, then 4, 5 and 6 would all stop at the first test and print `more than 3`. When the conditions overlap, put the most specific one first.

3.8.3 `switch`: choosing among many values

An `if` chain is fine for a few cases, but when one variable is compared against many specific values it becomes long and repetitive. The `switch` statement is made for this situation. It takes a variable of certain types (`byte`, `short`, `int`, `char`, and also `String` and enumerations) and jumps to the `case` matching its value.

The classic form of `switch` has a trap, and **Listing 3.35** walks straight into it first.

Listing 3.35 `SwitchDemo.java`: fall-through without `break`, the fix with `break` and `default`, and the arrow form.

```
public class SwitchDemo {

    public static void main(String[] args) {
        char input = 'b';

        System.out.println("-- without break --");
        switch (input) {
            case 'a': {
                System.out.println("Input was a");
            }
            case 'b': {
                System.out.println("Input was b");
            }
            case 'c': {
                System.out.println("Input was c");
            }
        }

        System.out.println("-- with break and default --");
        char[] tries = {'b', 'z'};
        for (char c : tries) {
            switch (c) {
                case 'a': {
                    System.out.println("Input was a");
                    break;
                }
                case 'b': {
                    System.out.println("Input was b");
                    break;
                }
                case 'c': {
                    System.out.println("Input was c");
                    break;
                }
                default: {
                    System.out.println("Unexpected letter");
                }
            }
        }
```

Listing 3.35 (Continued) *SwitchDemo.java: fall-through without break, the fix with break and default, and the arrow form.*

```
        System.out.println("-- arrow form (Java 14+) --");
        String code = "EXP";
        String label = switch (code) {
            case "REG" -> "Regular, 2 days";
            case "EXP" -> "Express, same day";
            default -> "Unknown service";
        };
        System.out.println(label);
    }
}
```

```
-- without break --
Input was b
Input was c
-- with break and default --
Input was b
Unexpected letter
-- arrow form (Java 14+) --
Express, same day
```

The first `switch` is meant to report which letter `input` holds. With `input = 'b'` it should print one line, but it prints two: `Input was b` *and* `Input was c`. The reason is that a `case` label is only an *entry point*. Once execution has jumped in at `case 'b':`, it simply continues downward through every statement that follows, including those under `case 'c':`, until it reaches the end of the `switch`. This is called **fall-through**, and **Figure 3.16** draws it.

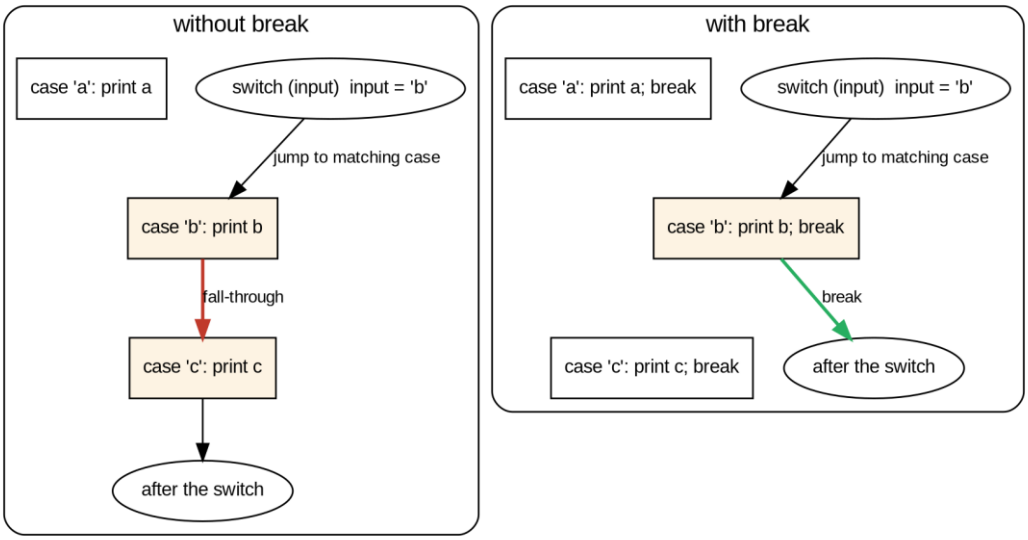


Figure 3.16 Without `break`, execution falls through from the matching case into the next; with `break`, it leaves the `switch`.

The cure is the keyword `break` at the end of each case, which jumps out of the `switch` immediately. The second `switch` in **Listing 3.35** adds `break` to every case, and it adds a **`default`** case, which runs when no case matches. The loop tries two letters: `'b'` prints only `Input was b`, and `'z'`, which matches nothing, prints `Unexpected letter`. The `default` case needs no `break`, because it is the last one and there is nothing to fall into.

Since Java 14, a second form of `switch` removes the trap entirely [9]. In the **arrow form**, `case "EXP" -> "Express, same day";`, each case is a single expression or block, there is no fall-through and no `break` to forget, and the whole `switch` produces a value that can be assigned. The third part of **Listing 3.35** uses it to turn a service code into a label. Prefer the arrow form in new code; learn the classic form because you will meet it in older programs.

Python has no fall-through at all. Since version 3.10 it has the `match` statement [10], whose `case _:` plays the role of `default`, and for simple lookups Python programmers often use a dictionary instead, as **Listing 3.36** shows. Dictionaries return in Chapter 9.

Listing 3.36 `switch_demo.py`: `match` (Python 3.10 and later), an `elif` chain, and a dictionary lookup.

```
def describe(letter):
    match letter:  # Python 3.10+; a match never falls through
        case "a":
            return "Input was a"
        case "b":
            return "Input was b"
        case "c":
            return "Input was c"
        case _:
            return "Unexpected letter"


def main():
    for c in ["b", "z"]:
        print(describe(c))

    code = "EXP"
    if code == "REG":
        label = "Regular, 2 days"
    elif code == "EXP":
        label = "Express, same day"
    else:
        label = "Unknown service"
    print(label)

    labels = {"REG": "Regular, 2 days", "EXP": "Express, same day"}
    print(labels.get("XYZ", "Unknown service"))


if __name__ == "__main__":
    main()
```

```
Input was b
Unexpected letter
Express, same day
Unknown service
```

When should you use `switch` rather than `if`? Use `switch` (or `match`) when every branch compares *one* variable against *specific constant values*. Use `if`/`else if` when the branches test ranges (`k > 4`), several different variables, or arbitrary conditions.

3.9 Loops

3.9.1 `while` and `for`

Sometimes we want to do something several times. We could write

```
System.out.println("Hello");
System.out.println("Hello");
System.out.println("Hello");
System.out.println("Hello");
```

but that is tedious, and impossible when the count is only known at run time. A **loop** says “do this, four times”; the two main kinds are `while` and `for`.

Here is a first attempt at both. Both are meant to print `Hello` four times, but one of them is not right. Which, and why?

```
int i = 0;
while (i < 4) {
    System.out.println("Hello");
}

for (int j = 0; j < 4; j++) {
    System.out.println("Hello");
}
```

The `for` loop is correct. The `while` loop never stops. Its condition `i < 4` is checked before each repetition, and nothing in the body ever changes `i`, so `i` stays 0, the condition stays true, and the program prints `Hello` for ever (until you stop it with Ctrl+C, or with the stop button in your editor). Such a loop is called an **infinite loop**. The fix is one line, `i++;`, at the end of the body. **Listing 3.37** contains the corrected loops, each printing its counter so you can see it advance.

Listing 3.37 `LoopBasics.java`: a corrected `while` loop, the equivalent `for` loop, and a `do-while` loop.

```
public class LoopBasics {

    public static void main(String[] args) {
        int i = 0;
        while (i < 4) {
            System.out.println("Hello (while, i = " + i + ")");
            i++;                        // without this line the loop never ends
        }

        for (int j = 0; j < 4; j++) {
            System.out.println("Hello (for, j = " + j + ")");
        }

        int k = 10;
        do {
            System.out.println("do-while runs once even though k = " + k);
        } while (k < 4);
    }
}
```

```
Hello (while, i = 0)
Hello (while, i = 1)
Hello (while, i = 2)
Hello (while, i = 3)
Hello (for, j = 0)
Hello (for, j = 1)
Hello (for, j = 2)
Hello (for, j = 3)
do-while runs once even though k = 10
```

A **`while` loop** has only a condition. Before every repetition, called an **iteration**, it checks the condition; if the condition is true, it runs the body and checks again; as soon as the condition is false, the loop ends and the program continues after it. If the condition is false the very first time, the body runs zero times. The important responsibility is yours: something in the body must eventually make the condition false.

A **`for` loop** puts all the bookkeeping in one line, in three parts separated by semicolons: `for (initialization; condition; update)`. **Figure 3.17** shows the order in which the parts run.

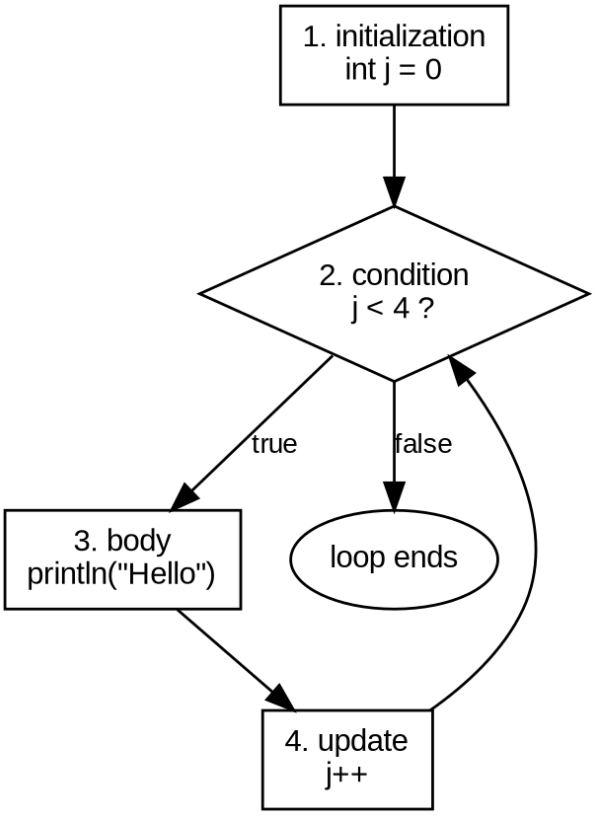


Figure 3.17 The four parts of `for (int j = 0; j < 4; j++)`: initialization once, then condition, body and update in a cycle until the condition is false.

1. The **initialization**, `int j = 0`, runs once, before anything else.
2. The **condition**, `j < 4`, is checked before each iteration; if it is false, the loop ends.
3. The **body** runs.
4. The **update**, `j++`, runs after the body, and control returns to step 2.

So `j` takes the values 0, 1, 2 and 3 in the body. With `for`, one line shows exactly how many times the body runs; with `while`, the counting is scattered over three places, and it is easy to forget one. Use `for` when you know the number of iterations, and `while` when you loop *until something happens*, such as "until the balance has doubled" in Exercise 3.8.

The third loop in **Listing 3.37** is a **`do-while` loop**. It checks its condition *after* the body, so the body always runs at least once, even here, where the condition `k < 4` is false from the start. It suits "ask, then check" situations, such as asking a user for input until the input is valid.

Python's loops, in **Listing 3.38**, look different but do the same.

Listing 3.38 `loop_basics.py`: Python's `while` loop, a `for` loop over `range`, and the `while True` pattern that replaces `do-while`.

```
def main():
    i = 0
    while i < 4:
        print("Hello (while, i = " + str(i) + ")")
        i += 1  # without this line the loop never ends

    for j in range(4):
        print("Hello (for, j = " + str(j) + ")")

    k = 10
    while True:  # Python has no do-while; this is the usual pattern
        print("do-while style runs once even though k = " + str(k))
        if not k < 4:
            break


if __name__ == "__main__":
    main()
```

```
Hello (while, i = 0)
Hello (while, i = 1)
Hello (while, i = 2)
Hello (while, i = 3)
Hello (for, j = 0)
Hello (for, j = 1)
Hello (for, j = 2)
Hello (for, j = 3)
do-while style runs once even though k = 10
```

Python's `while` is the same as Java's. Python's `for` always walks through the items of a sequence; to count, you ask for a sequence of numbers with **`range`**: `range(4)` produces 0, 1, 2, 3; `range(1, 11)` produces 1 up to 10 (the end is excluded); `range(10, 0, -1)` counts down from 10 to 1; and `range(3, 16, 3)` gives 3, 6, 9, 12, 15. Python has no `do-while`; the usual replacement is `while True:` with a `break`. In both languages `break` leaves a loop immediately, and `continue` skips the rest of the current iteration.

3.9.2 Four sequences

Here are four small loop problems. Try each on paper before looking at the solution. How would you write a `for` loop that prints

1. the numbers 1 to 10 in ascending order?
2. the numbers 1 to 10 in descending order?
3. the numbers 3, 6, 9, 12, 15?
4. the square numbers 1, 4, 9, 16, 25, …, 100, that is, the squares up to 10^2?

Each answer is a `for` loop whose three parts are chosen with care, in **Listing 3.39** and **Listing 3.40**.

Listing 3.39 `LoopSequences.java`: four `for` loops, one for each sequence.

```java
public class LoopSequences {

    public static void main(String[] args) {
        // 1-10, ascending
        for (int i = 1; i <= 10; i++) {
            System.out.print(i + " ");
        }
        System.out.println();

        // 1-10, descending
        for (int i = 10; i >= 1; i--) {
            System.out.print(i + " ");
        }
        System.out.println();

        // 3, 6, 9, 12, 15
        for (int i = 3; i <= 15; i += 3) {
            System.out.print(i + " ");
        }
        System.out.println();

        // 1, 4, 9, ..., 100: the squares of 1..10
        for (int i = 1; i <= 10; i++) {
            System.out.print(i * i + " ");
        }
        System.out.println();
    }
}
```

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
3 6 9 12 15
1 4 9 16 25 36 49 64 81 100
```

Listing 3.40 `loop_sequences.py`: the same sequences with `range`.

```python
def main():
    # 1-10, ascending
    for i in range(1, 11):
        print(i, end=" ")
    print()

    # 1-10, descending
    for i in range(10, 0, -1):
        print(i, end=" ")
    print()

    # 3, 6, 9, 12, 15
    for i in range(3, 16, 3):
        print(i, end=" ")
    print()

    # 1, 4, 9, ..., 100: the squares of 1..10
    for i in range(1, 11):
        print(i * i, end=" ")
    print()


if __name__ == "__main__":
    main()
```

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
3 6 9 12 15
1 4 9 16 25 36 49 64 81 100
```

Let us reason through each one.

1. **Ascending.** Start at 1, continue while `i <= 10`, add 1 each time. Writing `i < 10` would stop at 9; one iteration too many or too few is so common it has a name, the **off-by-one error**. In Python, `range(1, 11)` needs 11 because the end is excluded.
2. **Descending.** Start at 10, continue while `i >= 1`, and *subtract* 1 with `i--`. In Python the step is `-1`: `range(10, 0, -1)` stops before 0.
3. **Multiples of three.** Start at 3 and add 3 each time with `i += 3`; stop after 15 with `i <= 15`. An equally good alternative is to count `i` from 1 to 5 and print `3 * i`.
4. **Squares.** Rather than hunt for a clever update rule, count `i` from 1 to 10 and print `i * i`. The loop counter need not be the number printed; it only has to *determine* it.

`System.out.print`, without `ln`, prints without ending the line, so the numbers appear side by side; the empty `System.out.println()` after each loop ends the line. Python's `print(i, end=" ")` does the same by replacing the usual line ending with a space.

3.9.3 The for-each loop

Very often a loop only needs to visit every element of a collection, one after another, without caring about positions. Both languages have a loop for exactly that, the **for-each** loop. In Java it is written `for (String word : args)`, read aloud as "for each `word` in `args`". In Python it is simply the ordinary `for word in args:`. **Listing 3.41** prints the command-line words twice, first with an index-counting loop and then with a for-each loop, and **Listing 3.42** does the same in Python.

Listing 3.41 `ArgsEcho.java`: visiting every command-line word by index and with a for-each loop.

```java
public class ArgsEcho {

    public static void main(String[] args) {
        System.out.println("You gave " + args.length + " word(s).");
        for (int i = 0; i < args.length; i++) {
            System.out.println("args[" + i + "] = " + args[i]);
        }
        for (String word : args) {            // for-each: no index needed
            System.out.print(word.toUpperCase() + " ");
        }
        System.out.println();
    }
}
```

Run with three words, `java ArgsEcho wash dry iron`:

```
You gave 3 word(s).
args[0] = wash
args[1] = dry
args[2] = iron
WASH DRY IRON
```

Listing 3.42 `args_echo.py`: the same in Python, with `enumerate` for index and value together.

```python
import sys


def main():
    args = sys.argv[1:]  # sys.argv[0] is the script name itself
    print("You gave " + str(len(args)) + " word(s).")
    for i in range(len(args)):
        print("args[" + str(i) + "] = " + args[i])
    for i, word in enumerate(args):  # index and value together
        print(str(i) + ":" + word, end=" ")
    print()
    for word in args:  # for-each: no index needed
        print(word.upper(), end=" ")
    print()


if __name__ == "__main__":
    main()
```

Run as `python3 args_echo.py wash dry iron`:

```
You gave 3 word(s).
args[0] = wash
args[1] = dry
args[2] = iron
0:wash 1:dry 2:iron
WASH DRY IRON
```

The for-each loop is shorter and has no index to get wrong; use it whenever you do not need the position. When you need both, use the index loop in Java or `enumerate` in Python. Arrays and lists are the subject of Chapter 5.

3.9.4 Choosing the right loop

Table 3.10 The loop forms of Java and Python, and when to use each.

Loop	Checks condition	Runs at least once?	Best when
`for (init; cond; update)` / `for i in range(...)`	before each iteration	no	you know how many iterations
`while (cond)` / `while cond:`	before each iteration	no	you loop until something happens
`do { } while (cond);` / `while True: ... break`	after each iteration	yes	you must act before you can check
`for (T x : items)` / `for x in items:`	(visits each item)	no (if empty)	you visit every element, no index needed

Table 3.10 summarizes the choice. Any loop can be rewritten as any other, so the choice is not about power but about clarity: pick the loop whose shape tells the reader, at a glance, what kind of repetition this is.

3.10 Tracing Tables

How do you *know* what a loop does? Running it tells you what happens for one input, and nothing about *why* a wrong output is wrong. The professional technique is to **trace** the loop by hand with a **tracing table**: one column for each variable, one row for each moment you want to record, usually the end of each iteration. You fill it in with a pencil, playing the computer, one statement at a time [11]. It is the debugger of Chapter 2 carried in your head.

Let us trace **Listing 3.43** (in Python, **Listing 3.44**). It takes the number 4071 and computes two things at once: the sum of its digits and the number with its digits reversed.

Listing 3.43 `TraceDemo.java`: a `while` loop that takes a number apart digit by digit.

```
public class TraceDemo {

    public static void main(String[] args) {
        int n = 4071;
        int digitSum = 0;
        int reversed = 0;
        while (n > 0) {
            int digit = n % 10;              // last digit
            digitSum = digitSum + digit;
            reversed = reversed * 10 + digit;
            n = n / 10;                      // drop the last digit
        }
        System.out.println("digit sum = " + digitSum);
        System.out.println("reversed  = " + reversed);
    }
}
```

```
digit sum = 12
reversed  = 1704
```

Listing 3.44 `trace_demo.py`: the same loop in Python, with `//` for integer division.

```
def main():
    n = 4071
    digit_sum = 0
    reversed_number = 0
    while n > 0:
        digit = n % 10  # last digit
        digit_sum = digit_sum + digit
        reversed_number = reversed_number * 10 + digit
        n = n // 10  # drop the last digit
    print("digit sum = " + str(digit_sum))
    print("reversed  = " + str(reversed_number))


if __name__ == "__main__":
    main()
```

```
digit sum = 12
reversed  = 1704
```

The loop uses the two tools of Section 3.7.4. `n % 10` is the last digit of `n`, and `n / 10` (in Python `n // 10`) is `n` with its last digit removed. Before the loop, `n = 4071`, `digitSum = 0`, `reversed = 0`. **Table 3.11** records the state after each iteration.

Table 3.11 Tracing **Listing 3.43** for $n = 4071$; the loop ends when n reaches 0.

Iteration	n at start	digit = n % 10	digitSum	reversed	n = n / 10	n > 0?
(before)	4071		0	0		true
1	4071	1	0 + 1 = 1	0 · 10 + 1 = 1	407	true
2	407	7	1 + 7 = 8	1 · 10 + 7 = 17	40	true
3	40	0	8 + 0 = 8	17 · 10 + 0 = 170	4	true
4	4	4	8 + 4 = 12	170 · 10 + 4 = 1704	0	false

The table shows why the output is `12` and `1704`, and more: the loop runs once per digit, because each iteration removes one, and the trailing zero of 4071 becomes a *leading* zero of the reversal, which disappears, so the reverse is 1704, not 17040.

Tracing tables have a few rules that keep them honest.

1. Make one column per variable that changes, plus one for the loop condition.
2. Write the starting values in a first row before the loop.
3. Execute the body one statement at a time, *in order*, using the values as they are at that moment, not as they will be at the end of the iteration.
4. Record the state at the same point in every iteration, and check the condition at the point where the loop checks it.
5. Stop when the condition is false, and read the results from the last row.

Rule 3 is the one people break. In iteration 2, `reversed = reversed * 10 + digit` uses the `reversed` from iteration 1 (which is 1) and the `digit` computed a moment earlier in the same iteration (7). Get the order wrong and the table lies.

Section 3.11 and the exercises use tracing tables again, and Chapter 4 works whole problems of exactly this kind.

3.11 Worked Exercises, Step by Step

The three problems in this section use everything so far. For each we understand the problem, choose the types, write the program in both languages, and check the output by hand, the step that turns “the program printed something” into “the program is right”.

3.11.1 The area of a circle

Problem. The area A of a circle with radius r is

$$A = \pi r^2 \tag{3.5}$$

Write a program that computes it, with the variable `r` initialized to 5, and prints the result. Which types should the variables `A` and `r` have?

Choosing the types. A radius may well be a decimal, such as 2.5 cm, and the area certainly will be, because $\pi = 3.14159\ldots$ is not a whole number. So both variables are `double`. An `int` area would be wrong: Java refuses to store the `double` product in it without a cast, and a cast would throw away the fraction. Java’s `Math.PI` and Python’s `math.pi` provide π to full `double` precision. Equation (3.5) is then one line of code, `Math.PI * r * r`; multiplying the radius by itself is simpler and more accurate than `Math.pow(r, 2)`.

Listing 3.45 and **Listing 3.46** give the solution. The documentation comment at its start explains the choices, as a good solution should. Java variable names begin with a lowercase letter, so the area is called `a`.

Listing 3.45 `CircleArea.java`: the area of a circle with radius 5.

```java
public class CircleArea {
    /** The variables a and r are declared as double; a is computed
     *  from r as Math.PI * r * r and printed suitably.
     *  The program is tested with r = 5.
     */
    public static void main(String[] args) {
        double a;
        double r;
        r = 5;
        a = Math.PI * r * r;
        System.out.println("The area of a circle with radius " + r
                + " is " + a + ".");
        System.out.printf("Rounded to two decimals: %.2f%n", a);
    }
}
```

```
The area of a circle with radius 5.0 is 78.53981633974483.
Rounded to two decimals: 78.54
```

Listing 3.46 `circle_area.py`: the same computation in Python.

```python
import math


def main():
    """a and r are floats; a is computed from r as math.pi * r * r
    and printed suitably. The program is tested with r = 5.
    """
    r = 5.0
    a = math.pi * r * r
    print("The area of a circle with radius " + str(r)
          + " is " + str(a) + ".")
    print(f"Rounded to two decimals: {a:.2f}")


if __name__ == "__main__":
    main()
```

```
The area of a circle with radius 5.0 is 78.53981633974483.
Rounded to two decimals: 78.54
```

Checking by hand. $r^2 = 25$ and $25 \times 3.14159265\ldots = 78.5398\ldots$, which matches. The raw output shows sixteen significant digits, far more than anyone needs, which is why the second line formats the area to two decimals with `%.2f`.

3.11.2 From imperial units to kilograms

Problem. Write a program that converts a mass given in the imperial system into kilograms, using the conversions in **Table 3.12**. A person's weight is 11 stones and 6 pounds; store it as `stones = 11` and `pounds = 6`, with all the other units 0. How many kilograms is this?

Table 3.12 The imperial units of mass and their value in pounds.

Unit	In pounds
1 ton	2240 pounds
1 hundredweight	112 pounds
1 quarter	28 pounds
1 stone	14 pounds
1 ounce	$\frac{1}{16}$ pound
1 drachm	$\frac{1}{256}$ pound
1 grain	$\frac{1}{7000}$ pound
1 pound	0.45359237 kilograms

Planning. The plan has two stages. First convert every unit to pounds and add them up; then convert pounds to kilograms. Writing t, h, q, s, p, o, d, g for the numbers of tons, hundredweights, quarters, stones, pounds, ounces, drachms and grains, the weight in pounds is

$$W_{\text{lb}} = 2240t + 112h + 28q + 14s + p + \frac{o}{16} + \frac{d}{256} + \frac{g}{7000} \tag{3.6}$$

and the weight in kilograms is

$$W_{\text{kg}} = 0.45359237 \cdot W_{\text{lb}} \tag{3.7}$$

Rather than scattering the numbers 2240, 112, 28 and so on through the formula, the program first stores each conversion factor in a well-named variable such as `poundsPerStone`. A reader then sees `stones * poundsPerStone` and understands it immediately, and a mistyped factor has only one place to hide. Numbers that appear in code without a name are called **magic numbers**, and naming them is one of the cheapest improvements you can make to any program.

A trap in the factors. Look at how the fractions are written: `1 / 16.0`, not `1 / 16`. By Section 3.7.4, `1 / 16` is integer division and gives 0, which would silently make every ounce weigh nothing. Writing `16.0` makes it a decimal division. (In Python, `1 / 16` is already a decimal division.) The solution is **Listing 3.47** in Java and **Listing 3.48** in Python.

Listing 3.47 `ImperialToKg.java`: converting 11 stones and 6 pounds to kilograms.

```
public class ImperialToKg {
    /**
     * The conversion factors are stored in variables first. The imperial
     * weight is then turned into pounds, the pounds into kilograms, and
     * the program is run for 11 stones and 6 pounds.
     */
    public static void main(String[] args) {
        double poundsPerTon           = 2240;
        double poundsPerHundredweight = 112;
        double poundsPerQuarter       = 28;
        double poundsPerStone         = 14;
        double poundsPerOunce         = 1 / 16.0;
        double poundsPerDrachm        = 1 / 256.0;
        double poundsPerGrain         = 1 / 7000.0;
        double kilogramsPerPound      = 0.45359237;

        double tons = 0;
        double hundredweights = 0;
        double quarters = 0;
        double stones = 0;
        double pounds = 0;
        double ounces = 0;
        double drachms = 0;
        double grains = 0;

        /* First convert everything to pounds, then multiply
         * by kilogramsPerPound.
         */
        stones = 11;
        pounds = 6;
        double weightInPounds =
                tons * poundsPerTon
                + hundredweights * poundsPerHundredweight
                + quarters * poundsPerQuarter
                + stones * poundsPerStone
                + pounds
                + ounces * poundsPerOunce
                + drachms * poundsPerDrachm
                + grains * poundsPerGrain;
        double weightInKilograms = weightInPounds * kilogramsPerPound;
        System.out.println("A person with " + stones + " stones and " + pounds
                + " pounds has a weight corresponding to "
                + weightInKilograms + " kilograms.");
        System.out.printf("That is %.1f pounds, or about %.2f kg.%n",
                weightInPounds, weightInKilograms);
    }
```

Listing 3.47 (Continued) *ImperialToKg.java: converting 11 stones and 6 pounds to kilograms.*

```
}
```

```
A person with 11.0 stones and 6.0 pounds has a weight corresponding to 72.57477920000001
kilograms.
That is 160.0 pounds, or about 72.57 kg.
```

Listing 3.48 `imperial_to_kg.py`: the same conversion in Python.

```
def main():
    """The conversion factors are stored in variables first. The imperial
    weight is then turned into pounds, the pounds into kilograms, and the
    program is run for 11 stones and 6 pounds.
    """
    pounds_per_ton = 2240
    pounds_per_hundredweight = 112
    pounds_per_quarter = 28
    pounds_per_stone = 14
    pounds_per_ounce = 1 / 16
    pounds_per_drachm = 1 / 256
    pounds_per_grain = 1 / 7000
    kilograms_per_pound = 0.45359237

    tons = 0.0
    hundredweights = 0.0
    quarters = 0.0
    stones = 0.0
    pounds = 0.0
    ounces = 0.0
    drachms = 0.0
    grains = 0.0

    # First convert everything to pounds, then multiply by kilograms_per_pound.
    stones = 11.0
    pounds = 6.0
    weight_in_pounds = (tons * pounds_per_ton
                        + hundredweights * pounds_per_hundredweight
                        + quarters * pounds_per_quarter
                        + stones * pounds_per_stone
                        + pounds
                        + ounces * pounds_per_ounce
                        + drachms * pounds_per_drachm
                        + grains * pounds_per_grain)
    weight_in_kilograms = weight_in_pounds * kilograms_per_pound
    print("A person with " + str(stones) + " stones and " + str(pounds)
          + " pounds has a weight corresponding to "
          + str(weight_in_kilograms) + " kilograms.")
    print(f"That is {weight_in_pounds:.1f} pounds, or about {weight_in_kilograms:.2f} kg.")


if __name__ == "__main__":
    main()
```

```
A person with 11.0 stones and 6.0 pounds has a weight corresponding to 72.57477920000001
kilograms.
That is 160.0 pounds, or about 72.57 kg.
```

Checking by hand. Only two terms of Equation (3.6) are non-zero: $14 \times 11 = 154$ pounds from the stones, plus 6 pounds, gives $W_{\text{lb}} = 160$. By Equation (3.7), $W_{\text{kg}} = 160 \times 0.45359237 = 72.5747792$ kg. The program printed `72.57477920000001`: the same number with a tiny floating-point error in the sixteenth digit, exactly the effect of Section 3.7.3. A person weighing 11 stones 6 pounds weighs about 72.57 kg, as the formatted line says.

3.11.3 An investment over five hundred years

Problem. A capital of GBP 100 (one hundred pounds sterling) is invested at a fixed interest rate of 2.3% per year. The interest is added to the capital at the end of each year, so the next year's interest is earned on the larger sum; this is **compound interest**. Use the formula

$$\text{total} = \text{capital} \cdot (1 + 0.01 \cdot \text{rate})^{\text{years}} \tag{3.8}$$

to print the balance after each of the first 5 years and after 500 years.

Why the formula works. After one year the capital is multiplied by $1 + 0.023 = 1.023$: the original 100% plus 2.3% interest. The second year multiplies *that* by 1.023 again, and so on. After n years the capital has been multiplied by 1.023 exactly n times, which is 1.023^n, and that is Equation (3.8). The factor 0.01 turns the percentage 2.3 into the fraction 0.023.

A first solution. At this point we know variables, arithmetic and printing, but we pretend not to know loops yet. The result, **Listing 3.49** (and **Listing 3.50** in Python), is honest but repetitive: the same two statements appear six times, with only the number of years changed.

Listing 3.49 `Investment.java`: Equation (3.8) evaluated for six numbers of years, one after another.

```
public class Investment {
    /** The formula total = capital * (1 + 0.01 * interestRate)^years is
     *  written out once for each of the years 1, 2, 3, 4, 5 and 500.
     *  The code is repetitive on purpose: InvestmentLoop.java removes
     *  the repetition with a loop.
     */
    public static void main(String[] args) {
        double capital = 100;       // the initial capital in pounds
        double interestRate = 2.3;  // the interest rate per year, in percent
        int years = 1;              // how long the capital is invested
        double total;               // the capital after interest is added

        total = capital * Math.pow(1 + 0.01 * interestRate, years);
        System.out.println("After " + years + " years a capital of GBP " + capital
                + " grows with an interest rate of " + interestRate + "\u0025 to GBP "
                + Math.round(total));
        years = 2;
        total = capital * Math.pow(1 + 0.01 * interestRate, years);
        System.out.println("After " + years + " years a capital of GBP " + capital
                + " grows with an interest rate of " + interestRate + "\u0025 to GBP "
                + Math.round(total));
        years = 3;
        total = capital * Math.pow(1 + 0.01 * interestRate, years);
        System.out.println("After " + years + " years a capital of GBP " + capital
                + " grows with an interest rate of " + interestRate + "\u0025 to GBP "
                + Math.round(total));
        years = 4;
        total = capital * Math.pow(1 + 0.01 * interestRate, years);
        System.out.println("After " + years + " years a capital of GBP " + capital
                + " grows with an interest rate of " + interestRate + "\u0025 to GBP "
                + Math.round(total));
        years = 5;
        total = capital * Math.pow(1 + 0.01 * interestRate, years);
        System.out.println("After " + years + " years a capital of GBP " + capital
                + " grows with an interest rate of " + interestRate + "\u0025 to GBP "
                + Math.round(total));
        years = 500;
        total = capital * Math.pow(1 + 0.01 * interestRate, years);
        System.out.println("After " + years + " years a capital of GBP " + capital
                + " grows with an interest rate of " + interestRate + "\u0025 to GBP "
                + Math.round(total));
    }
}
```

```
After 1 years a capital of GBP 100.0 grows with an interest rate of 2.3% to GBP 102
After 2 years a capital of GBP 100.0 grows with an interest rate of 2.3% to GBP 105
After 3 years a capital of GBP 100.0 grows with an interest rate of 2.3% to GBP 107
After 4 years a capital of GBP 100.0 grows with an interest rate of 2.3% to GBP 110
After 5 years a capital of GBP 100.0 grows with an interest rate of 2.3% to GBP 112
After 500 years a capital of GBP 100.0 grows with an interest rate of 2.3% to GBP 8665964
```

Listing 3.50 `investment.py`: the same repetitive solution in Python.

```
def main():
    """The formula total = capital * (1 + 0.01 * interest_rate) ** years is
    written out once for each of the years 1, 2, 3, 4, 5 and 500.
    The code is repetitive on purpose: investment_loop.py removes the
    repetition with a loop.
    """
    capital = 100.0      # the initial capital in pounds
    interest_rate = 2.3  # the interest rate per year, in percent
    years = 1            # how long the capital is invested

    total = capital * (1 + 0.01 * interest_rate) ** years
    print("After " + str(years) + " years a capital of GBP " + str(capital)
          + " grows with an interest rate of " + str(interest_rate) + "% to GBP "
          + str(round(total)))
    years = 2
    total = capital * (1 + 0.01 * interest_rate) ** years
    print("After " + str(years) + " years a capital of GBP " + str(capital)
          + " grows with an interest rate of " + str(interest_rate) + "% to GBP "
          + str(round(total)))
    years = 3
    total = capital * (1 + 0.01 * interest_rate) ** years
    print("After " + str(years) + " years a capital of GBP " + str(capital)
          + " grows with an interest rate of " + str(interest_rate) + "% to GBP "
          + str(round(total)))
    years = 4
    total = capital * (1 + 0.01 * interest_rate) ** years
    print("After " + str(years) + " years a capital of GBP " + str(capital)
          + " grows with an interest rate of " + str(interest_rate) + "% to GBP "
          + str(round(total)))
    years = 5
    total = capital * (1 + 0.01 * interest_rate) ** years
    print("After " + str(years) + " years a capital of GBP " + str(capital)
          + " grows with an interest rate of " + str(interest_rate) + "% to GBP "
          + str(round(total)))
    years = 500
    total = capital * (1 + 0.01 * interest_rate) ** years
    print("After " + str(years) + " years a capital of GBP " + str(capital)
          + " grows with an interest rate of " + str(interest_rate) + "% to GBP "
          + str(round(total)))


if __name__ == "__main__":
    main()
```

```
After 1 years a capital of GBP 100.0 grows with an interest rate of 2.3% to GBP 102
After 2 years a capital of GBP 100.0 grows with an interest rate of 2.3% to GBP 105
After 3 years a capital of GBP 100.0 grows with an interest rate of 2.3% to GBP 107
After 4 years a capital of GBP 100.0 grows with an interest rate of 2.3% to GBP 110
After 5 years a capital of GBP 100.0 grows with an interest rate of 2.3% to GBP 112
After 500 years a capital of GBP 100.0 grows with an interest rate of 2.3% to GBP 8665964
```

`Math.pow` computes the power (Python writes `**`), and `Math.round` rounds to the nearest whole pound. The percent sign is written `\u0025`, its Unicode escape; the compiler turns the escape into an ordinary `%` before it reads anything else, so a plain `%` would work just as well here. (In a format string, either spelling would have to be doubled to `%%`, Section 3.5.4.) The message “After 1 years” is ungrammatical; the loop version below avoids it with a table layout.

Checking by hand. **Table 3.13** works through the first five years.

Table 3.13 Evaluating Equation (3.8) by hand for the first five years.

Years n	1.023^n	$100 \cdot 1.023^n$	Rounded
1	1.023000	102.30	102
2	1.046529	104.65	105
3	1.070599	107.06	107
4	1.095223	109.52	110
5	1.120413	112.04	112

The rounded column matches the program. The last line is the startling one: after 500 years, GBP 100 has grown to more than GBP 8.6 million. That is the power of compounding, and **Figure 3.18** shows it. On a logarithmic vertical axis, where each step up multiplies by ten, the growth is a straight line, because the balance is multiplied by the same factor every year.

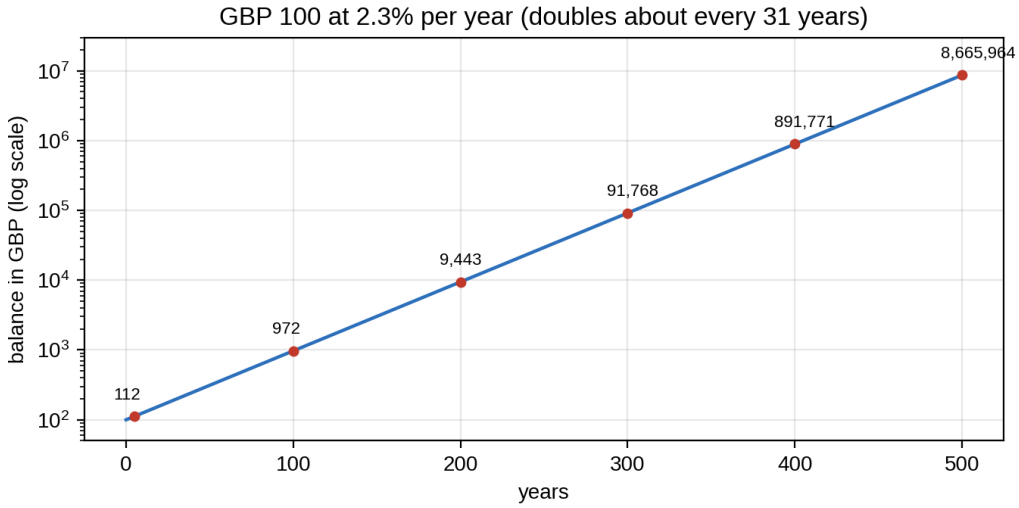


Figure 3.18 The balance of GBP 100 at 2.3% per year over 500 years, on a logarithmic scale; the straight line is the signature of compound growth.

Removing the repetition. Repeated code is a warning sign: if the message has a typo, it has six typos, and fixing five of them is a bug waiting to happen. A loop removes the repetition. **Listing 3.51** computes the balance year by year, multiplying by the factor once per iteration, and prints only the years we want: the first five and every hundredth. At the end it checks the loop against the formula. **Listing 3.52** is the Python version.

Listing 3.51 `InvestmentLoop.java`: the balance computed year by year in a loop.

```java
public class InvestmentLoop {

    public static void main(String[] args) {
        double capital = 100;       // GBP
        double interestRate = 2.3;  // percent per year
        double factor = 1 + 0.01 * interestRate;

        System.out.println("Year      Balance (GBP)");
        double balance = capital;
        for (int year = 1; year <= 500; year++) {
            balance = balance * factor;          // add one year of interest
            if (year <= 5 || year % 100 == 0) {
                System.out.printf("%4d %18.2f%n", year, balance);
            }
        }
        double direct = capital * Math.pow(factor, 500);
        System.out.printf("Formula check for year 500: %.2f%n", direct);
    }
}
```

```
Year      Balance (GBP)
   1             102.30
   2             104.65
   3             107.06
   4             109.52
   5             112.04
 100             971.77
 200            9443.36
 300           91767.76
 400          891771.29
 500         8665963.51
Formula check for year 500: 8665963.51
```

Listing 3.52 `investment_loop.py`: the loop version in Python.

```python
def main():
    capital = 100.0      # GBP
    interest_rate = 2.3  # percent per year
    factor = 1 + 0.01 * interest_rate

    print("Year      Balance (GBP)")
    balance = capital
    for year in range(1, 501):
        balance = balance * factor  # add one year of interest
        if year <= 5 or year % 100 == 0:
            print(f"{year:4d} {balance:18.2f}")
    direct = capital * factor ** 500
    print(f"Formula check for year 500: {direct:.2f}")


if __name__ == "__main__":
    main()
```

```
Year      Balance (GBP)
   1             102.30
   2             104.65
   3             107.06
   4             109.52
   5             112.04
 100             971.77
 200            9443.36
 300           91767.76
 400          891771.29
 500         8665963.51
Formula check for year 500: 8665963.51
```

The condition `year <= 5 || year % 100 == 0` combines Section 3.6.3 and Section 3.7.4: print when the year is one of the first five, or when it divides by 100 with no remainder. And the two ways of computing year 500, 500 multiplications in the loop and one `Math.pow`, agree to the cent, even though each multiplication carries a tiny rounding error; with `double` the errors stay far below one cent.

3.12 Profitina Laundry: Customer and Service

Profitina Laundry is a small neighborhood laundry. Customers bring a bag of clothes to the counter; the clerk weighs it, the customer chooses a service, and the clerk writes a receipt. There are three services: *Wash & Fold* at Rp7,000 per kilogram, *Iron Only* at Rp5,000 per kilogram, and *Wash, Dry & Iron* at Rp10,000 per kilogram. Each service has a minimum weight: a bag lighter than the minimum is charged as if it weighed the minimum. Weights are charged in steps of half a kilogram, always rounded up. Express service, finished the same day, costs 50% more. Registered members get 10% off.

These rules are typical business logic: simple each, fiddly together, and easy to get subtly wrong.

3.12.1 Finding the classes

The nouns in the description suggest the classes: *customer*, *service*, *bag*, *receipt*. For this chapter we build the first two; orders and receipts get their own classes once we have collections (Chapter 5).

A `Customer` has a name, a phone number, a membership flag and a count of visits. The name and phone number never change after registration, so they get getters but no setters, and in Java they are marked `final`, which makes the compiler refuse any second assignment. Membership can change, so it has a setter. The visit count changes only by one at a time, so instead of a setter there is a method `recordVisit()`. There is also `maskedPhone()`, which shows only the last three digits; Section 3.13 explains why.

A `Service` has a code, a name, a price per kilogram and a minimum weight. It knows how to compute the chargeable weight and the price of a bag. **Figure 3.19** shows the two classes and the demonstration program that uses them.

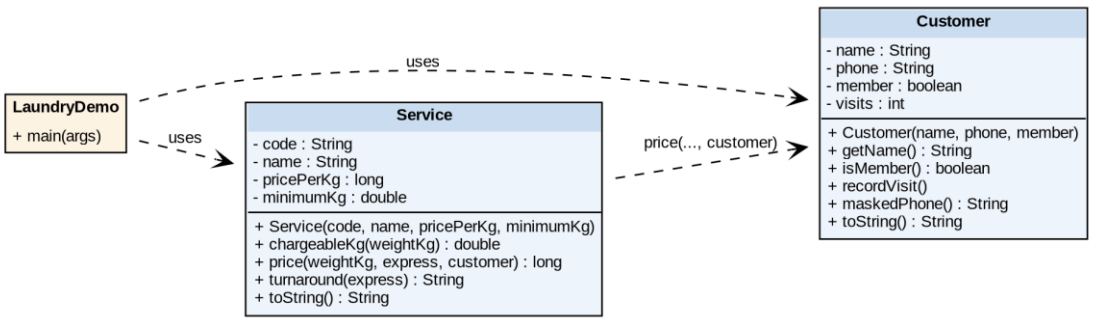


Figure 3.19 The first two classes of Profitina Laundry and the program that uses them.

3.12.2 The price rules as formulas

Before writing code, pin the rules down exactly. For a bag of weight w kilograms and a service with minimum m kilograms and price r rupiah per kilogram, the **chargeable weight** is the larger of w and m, rounded up to the next half kilogram:

$$c = \frac{\lceil 2 \cdot \max(w, m) \rceil}{2} \tag{3.9}$$

The brackets $\lceil \cdot \rceil$ mean "round up to a whole number" (the **ceiling**). Doubling first and halving after turns "round up to a half" into "round up to a whole": for $w = 4.2$, $2w = 8.4$, rounded up is 9, halved is 4.5. The base price is then $c \cdot r$. Express multiplies it by $\frac{3}{2}$, and membership subtracts a tenth, in that order:

$$p = \begin{cases} c \cdot r & \text{regular} \\ \frac{3}{2}\, c \cdot r & \text{express} \end{cases}, \qquad \text{price} = \begin{cases} p & \text{guest} \\ p - \lfloor p/10 \rfloor & \text{member} \end{cases} \tag{3.10}$$

The floor brackets $\lfloor \cdot \rfloor$ in Equation (3.10) mean "round down", which is exactly what integer division does. Because all prices are whole rupiah stored in integer types, as the warning in Section 3.7.3 recommends, every step after Equation (3.9) is exact integer arithmetic. **Figure 3.20** shows the calculation as a flowchart.

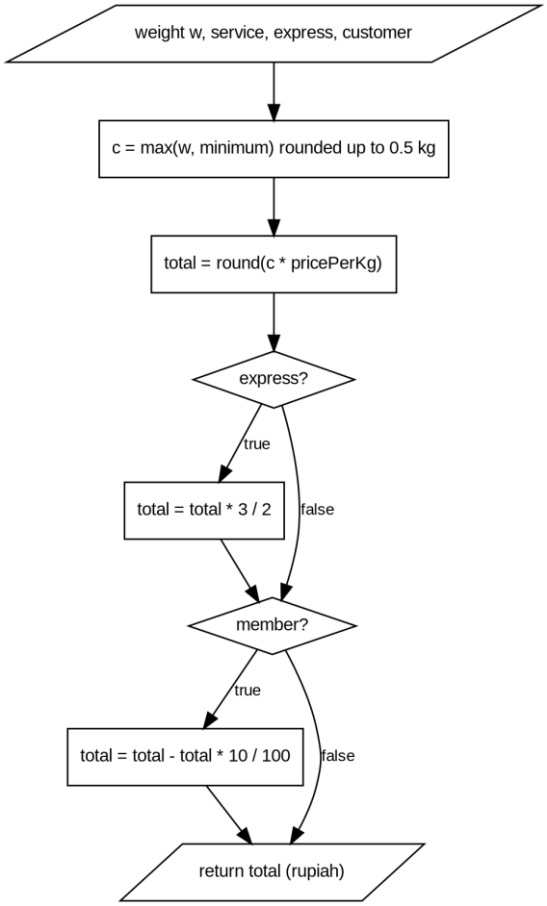


Figure 3.20 The price calculation of `Service.price`: chargeable weight, base price, then the two optional adjustments.

3.12.3 The code

Listing 3.53 and **Listing 3.54** are the Java classes.

Listing 3.53 `Customer.java`: a laundry customer with a masked phone number and a visit counter.

```
public class Customer {
    private final String name;
    private final String phone;
    private boolean member;
    private int visits;

    public Customer(String name, String phone, boolean member) {
        this.name = name;
        this.phone = phone;
        this.member = member;
        this.visits = 0;
    }

    public String getName() {
        return name;
    }

    public String getPhone() {
        return phone;
    }

    public boolean isMember() {
        return member;
    }

    public void setMember(boolean member) {
        this.member = member;
    }

    public int getVisits() {
        return visits;
    }

    public void recordVisit() {
        visits++;
    }

    /** Shows only the last three digits of the phone number. */
    public String maskedPhone() {
        int keep = 3;
        if (phone.length() <= keep) {
            return phone;
        }
        return "*".repeat(phone.length() - keep) + phone.substring(phone.length() - keep);
    }
```

Listing 3.53 (Continued) *Customer.java: a laundry customer with a masked phone number and a visit counter.*

```
    @Override
    public String toString() {
        String kind = member ? "member" : "guest";
        return name + " (" + kind + ", " + maskedPhone() + ", " + visits + " visit(s))";
    }
}
```

Listing 3.54 `Service.java`: a laundry service that computes the chargeable weight, the price, and the turnaround time.

```
public class Service {
    private final String code;
    private final String name;
    private final long pricePerKg;     // whole rupiah
    private final double minimumKg;

    public Service(String code, String name, long pricePerKg, double minimumKg) {
        this.code = code;
        this.name = name;
        this.pricePerKg = pricePerKg;
        this.minimumKg = minimumKg;
    }

    public String getCode() {
        return code;
    }

    public String getName() {
        return name;
    }

    public long getPricePerKg() {
        return pricePerKg;
    }

    public double getMinimumKg() {
        return minimumKg;
    }

    /** The weight that is charged: at least the minimum, rounded up to 0.5 kg. */
    public double chargeableKg(double weightKg) {
        double kg = Math.max(weightKg, minimumKg);
        return Math.ceil(kg * 2) / 2;
    }

    /** The price in rupiah for one bag of laundry. */
    public long price(double weightKg, boolean express, Customer customer) {
        long total = Math.round(chargeableKg(weightKg) * pricePerKg);
        if (express) {
            total = total * 3 / 2;            // express costs 50% more
        }
        if (customer.isMember()) {
            total = total - total * 10 / 100; // members get 10% off
        }
        return total;
    }
```

Listing 3.54 (Continued) *Service.java: a laundry service that computes the chargeable weight, the price, and the turnaround time.*

```
    public String turnaround(boolean express) {
        if (express) {
            return "same day";
        }
        return switch (code) {
            case "WF" -> "2 days";
            case "IR" -> "1 day";
            case "WDI" -> "3 days";
            default -> "ask at the counter";
        };
    }

    @Override
    public String toString() {
        return String.format("%-4s %-18s Rp%,6d/kg (min %.1f kg)", code, name, pricePerKg,
minimumKg);
    }
}
```

Every idea of this chapter appears in these two classes: private fields, constructors that use `this`, getters, and a setter only for the field that may change. `maskedPhone` uses `length`, `substring` and `repeat` behind an `if`. `chargeableKg` implements Equation (3.9) with `Math.max` and `Math.ceil`. `price` receives a `Customer` and asks it `customer.isMember()`, which is how objects cooperate, then applies Equation (3.10) in exact `long` arithmetic, multiplying before dividing in `total * 3 / 2` so that no fraction is lost. `turnaround` uses an early `return` and an arrow-form `switch`.

The demonstration program, **Listing 3.55**, registers two customers, processes four bags in a loop, and prints a small report. Arrays, the `{...}` lists of values, are covered properly in Chapter 5; here we use them only as four parallel columns, where position `i` in each array describes bag number `i`.

Listing 3.55 `LaundryDemo.java`: pricing four bags of laundry for two customers.

```
public class LaundryDemo {

    public static void main(String[] args) {
        Service washFold = new Service("WF", "Wash & Fold", 7000, 3.0);
        Service ironOnly = new Service("IR", "Iron Only", 5000, 2.0);
        Service full = new Service("WDI", "Wash, Dry & Iron", 10000, 3.0);

        System.out.println("PROFITINA LAUNDRY - price list");
        System.out.println(washFold);
        System.out.println(ironOnly);
        System.out.println(full);
        System.out.println();

        Customer siti = new Customer("Siti Aminah", "081234567890", true);
        Customer budi = new Customer("Budi Santoso", "085712340000", false);

        Customer[] who = {siti, budi, budi, siti};
        Service[] what = {washFold, full, ironOnly, washFold};
        double[] kg = {4.2, 2.0, 5.75, 12.0};
        boolean[] express = {false, true, false, false};

        long grandTotal = 0;
        for (int i = 0; i < who.length; i++) {
            who[i].recordVisit();
            long price = what[i].price(kg[i], express[i], who[i]);
            grandTotal += price;
            System.out.printf("%-13s %-18s %5.2f kg -> %4.1f kg  Rp%,8d  %s%n",
                    who[i].getName(), what[i].getName(), kg[i],
                    what[i].chargeableKg(kg[i]), price, what[i].turnaround(express[i]));
        }
        System.out.printf("Total for the day: Rp%,d%n", grandTotal);
        System.out.println(siti);
        System.out.println(budi);
    }
}
```

```
javac Customer.java Service.java LaundryDemo.java
java LaundryDemo
```

```
PROFITINA LAUNDRY - price list
WF   Wash & Fold        Rp 7,000/kg (min 3.0 kg)
IR   Iron Only          Rp 5,000/kg (min 2.0 kg)
WDI  Wash, Dry & Iron   Rp10,000/kg (min 3.0 kg)

Siti Aminah   Wash & Fold         4.20 kg ->  4.5 kg  Rp  28,350  2 days
Budi Santoso  Wash, Dry & Iron    2.00 kg ->  3.0 kg  Rp  45,000  same day
Budi Santoso  Iron Only           5.75 kg ->  6.0 kg  Rp  30,000  1 day
Siti Aminah   Wash & Fold        12.00 kg -> 12.0 kg  Rp  75,600  2 days
Total for the day: Rp178,950
Siti Aminah (member, *********890, 2 visit(s))
Budi Santoso (guest, *********000, 2 visit(s))
```

The Python classes follow the same design line for line, in **Listing 3.56**, **Listing 3.57** and **Listing 3.58**.

Listing 3.56 `customer.py`: the `Customer` class in Python.

```
class Customer:
    """A laundry customer: name, phone number, membership and visit count."""

    def __init__(self, name, phone, member):
        self._name = name
        self._phone = phone
        self._member = member
        self._visits = 0

    def get_name(self):
        return self._name

    def get_phone(self):
        return self._phone

    def is_member(self):
        return self._member

    def set_member(self, member):
        self._member = member

    def get_visits(self):
        return self._visits

    def record_visit(self):
        self._visits += 1

    def masked_phone(self):
        """Shows only the last three digits of the phone number."""
        keep = 3
        if len(self._phone) <= keep:
            return self._phone
        return "*" * (len(self._phone) - keep) + self._phone[-keep:]

    def __str__(self):
        kind = "member" if self._member else "guest"
        return (self._name + " (" + kind + ", " + self.masked_phone() + ", "
                + str(self._visits) + " visit(s))")
```

Listing 3.57 `service.py`: the `Service` class in Python.

```
import math


class Service:
    """A laundry service with a price per kilogram and a minimum weight."""

    def __init__(self, code, name, price_per_kg, minimum_kg):
        self._code = code
        self._name = name
        self._price_per_kg = price_per_kg  # whole rupiah
        self._minimum_kg = minimum_kg

    def get_code(self):
        return self._code

    def get_name(self):
        return self._name

    def get_price_per_kg(self):
        return self._price_per_kg

    def get_minimum_kg(self):
        return self._minimum_kg

    def chargeable_kg(self, weight_kg):
        """The weight that is charged: at least the minimum, rounded up to 0.5 kg."""
        kg = max(weight_kg, self._minimum_kg)
        return math.ceil(kg * 2) / 2

    def price(self, weight_kg, express, customer):
        """The price in rupiah for one bag of laundry."""
        total = int(self.chargeable_kg(weight_kg) * self._price_per_kg)
        if express:
            total = total * 3 // 2  # express costs 50% more
        if customer.is_member():
            total = total - total * 10 // 100  # members get 10% off
        return total
```

Listing 3.57 (Continued) *service.py: the Service class in Python.*

```
    def turnaround(self, express):
        if express:
            return "same day"
        match self._code:
            case "WF":
                return "2 days"
            case "IR":
                return "1 day"
            case "WDI":
                return "3 days"
            case _:
                return "ask at the counter"

    def __str__(self):
        return (f"{self._code:<4} {self._name:<18} Rp{self._price_per_kg:6,d}/kg "
                f"(min {self._minimum_kg:.1f} kg)")
```

Listing 3.58 `laundry_demo.py`: the demonstration program in Python.

```
from customer import Customer
from service import Service


def main():
    wash_fold = Service("WF", "Wash & Fold", 7000, 3.0)
    iron_only = Service("IR", "Iron Only", 5000, 2.0)
    full = Service("WDI", "Wash, Dry & Iron", 10000, 3.0)

    print("PROFITINA LAUNDRY - price list")
    print(wash_fold)
    print(iron_only)
    print(full)
    print()

    siti = Customer("Siti Aminah", "081234567890", True)
    budi = Customer("Budi Santoso", "085712340000", False)

    who = [siti, budi, budi, siti]
    what = [wash_fold, full, iron_only, wash_fold]
    kg = [4.2, 2.0, 5.75, 12.0]
    express = [False, True, False, False]

    grand_total = 0
    for i in range(len(who)):
        who[i].record_visit()
        price = what[i].price(kg[i], express[i], who[i])
        grand_total += price
        print(f"{who[i].get_name():<13} {what[i].get_name():<18} {kg[i]:5.2f} kg -> "
              f"{what[i].chargeable_kg(kg[i]):4.1f} kg  Rp{price:8,d}  "
              f"{what[i].turnaround(express[i])}")
    print(f"Total for the day: Rp{grand_total:,d}")
    print(siti)
    print(budi)


if __name__ == "__main__":
    main()
```

```
PROFITINA LAUNDRY - price list
WF   Wash & Fold        Rp 7,000/kg (min 3.0 kg)
IR   Iron Only          Rp 5,000/kg (min 2.0 kg)
WDI  Wash, Dry & Iron   Rp10,000/kg (min 3.0 kg)

Siti Aminah   Wash & Fold         4.20 kg ->  4.5 kg  Rp  28,350  2 days
Budi Santoso  Wash, Dry & Iron    2.00 kg ->  3.0 kg  Rp  45,000  same day
Budi Santoso  Iron Only           5.75 kg ->  6.0 kg  Rp  30,000  1 day
Siti Aminah   Wash & Fold        12.00 kg -> 12.0 kg  Rp  75,600  2 days
Total for the day: Rp178,950
Siti Aminah (member, *********890, 2 visit(s))
Budi Santoso (guest, *********000, 2 visit(s))
```

The two outputs are identical, character for character. Small translation choices made that possible: `math.ceil` for `Math.ceil`, `//` wherever Java's `/` divides integers, and `int(...)` instead of Python's `round`, which rounds halves to even. Since c is a multiple of 0.5 and every price per kilogram is a multiple of 1000, the product $c \cdot r$ is always a whole number, so `int` loses nothing here.

3.12.4 Checking the prices by hand

A price calculation that is wrong by a few thousand rupiah on every bag is a disaster for a small business, so we check every line of the output with a tracing table, **Table 3.14**.

Table 3.14 Checking the four prices of **Listing 3.55** with Equation (3.9) and Equation (3.10).

Bag	Service, r, m	w	c By Equation (3.9)	$c \cdot r$	Express	Member	Price
1 Siti	WF, 7000, 3.0	4.20	$\lceil 8.4 \rceil / 2 = 4.5$	31 500	no	−3 150	28 350
2 Budi	WDI, 10000, 3.0	2.00	$\lceil 6 \rceil / 2 = 3.0$	30 000	$\times \frac{3}{2} = 45\,000$	no	45 000
3 Budi	IR, 5000, 2.0	5.75	$\lceil 11.5 \rceil / 2 = 6.0$	30 000	no	no	30 000
4 Siti	WF, 7000, 3.0	12.00	$\lceil 24 \rceil / 2 = 12.0$	84 000	no	−8 400	75 600

Bag 2 shows the minimum at work: Budi's 2 kg are charged as 3 kg. Bag 3 shows the rounding: 5.75 kg is charged as 6 kg. The four prices add up to $28\,350 + 45\,000 + 30\,000 + 75\,600 = 178\,950$, the day's total in the output. The last two lines show the objects' state after the loop: each customer visited twice, and the phone numbers are masked.

3.13 Profitina in Practice

Profitina, the AI Visibility platform at app.profitina.com, measures how ready a brand's website is to be cited by AI answer engines, and then helps fix what it finds. Its server is a Python web application, and the ideas of this chapter show up in its structure.

Classes for the things of the domain. Just as the laundry has `Customer` and `Service`, the scanner models each kind of signal it examines (how extractable a page's content is, whether its structured data is valid, how technically sound the site is) as its own class that scores and explains itself. Chapter 6 shows the shared interface these classes implement.

Strings everywhere. A web page is text. Deciding whether a page mentions a brand, checking a `robots.txt` file, reading a schema.org block: all of it is string work with the methods of Section 3.5, and all of it compares *content*, never identity, the lesson of **Table 3.3**.

Numbers with care. Scores are decimals, compared against thresholds with the tolerance of Equation (3.3), never with `==`. Amounts of the PROFIT utility coin, which has six decimal places, are handled as whole numbers of the smallest unit, like the laundry's rupiah.

Personal data, handled deliberately. The laundry's `maskedPhone` method is not decoration. Indonesia's Personal Data Protection Law, UU PDP No. 27 of 2022 [12], requires that personal data be processed only as needed for its purpose. A receipt or a screen that shows `*********890` still lets a clerk confirm the customer, without exposing the full number to everyone at the counter. Profitina follows the same principle in its own compliance features, with explicit consent, data export and complete account deletion.

Tests as a gate. Profitina's release process runs well over a thousand automated tests of the same shape as `DogCheck`, expected value against actual value, and a release goes out only when all of them pass. The platform advises; its users decide.

3.14 Common Mistakes, and How to Avoid Them

Forgetting `this` in a constructor. `name = name;` assigns the parameter to itself. Write `this.name = name;`; in Python, `self.name = name`.

Comparing strings with `==` in Java. It compares arrows, not characters. Use `equals` (Section 3.5.3); in Python, `==` is right and `is` is wrong.

Expecting a string method to change the string. `name.trim();` alone throws the result away. Write `name = name.trim();`.

Believing an assignment copies an object. `Dog b = a;` makes an alias. Draw the object diagram (Section 3.4.3).

Calling a method through `null` or `None`. Check first, or make sure the variable is always given an object.

Integer overflow in the middle of a calculation. Cast to `long` *before* the risky multiplication, `(long) x * y`, not after it.

Comparing decimals with `==`. Use a tolerance, Equation (3.3), or `math.isclose` in Python.

Integer division by accident. `1 / 16` is 0 in Java: write `1 / 16.0` or `(double) x / y`, and remember that negative numbers divide differently in the two languages (Section 3.7.4).

Storing money in `double`. Use whole units in `long`, or `BigDecimal`/`Decimal`.

A missing `break` in a classic `switch`. Execution falls through into the next case. Prefer the arrow form in Java, or `match` in Python.

An `else if` chain in the wrong order. The first true condition wins; put the most specific condition first.

A `while` loop that never ends. Something in the body must move the condition toward false. Check it before you run the program.

Off-by-one errors. `<` versus `<=`, starting at 0 or 1, Python's excluded `range` end: trace the first and last iteration by hand.

Starting at `args[0]` in Python. `sys.argv[0]` is the script name; the first word is `sys.argv[1]`.

Comments that repeat the code. Explain why, not what.

3.15 Chapter Summary

- A **class** is a blueprint; an **object** (an **instance**) is one thing built from it, with its own copy of every **field**. **Constructors** build objects, **methods** give them behavior, and `this` (Java) or `self` (Python) means "the current object". Objects are made with `new` in Java and by calling the class in Python; private fields are reached through **getters** and **setters**, and `toString` or `__str__` gives the text form.
- Java variables of **primitive types** hold values; variables of **reference types** hold arrows to objects. In Python every variable is a name bound to an object. Copying a reference makes an **alias**. Identity is `==` in Java and `is` in Python; content equality is `equals` in Java and `==` in Python. **Object diagrams** make all of this visible.
- **Strings** are immutable, indexed from 0, and rich in methods (and slices in Python). Format output with `printf`/`String.format` or f-strings; parse text with `Integer.parseInt`, `Double.parseDouble`, `int` and `float`.
- Java is **statically typed**; Python is **dynamically** but **strongly typed**.
- Three arithmetic pitfalls: **overflow** (Equation (3.2)), **floating-point rounding** (compare with a tolerance, Equation (3.3)), and **integer division** (`1 / 16` is 0 in Java; Python separates `/` and `//`).
- **Conditionals**: `if`/`else if`/`else` (`elif`) tested from the top; `switch` (with `break`, or the arrow form) and `match` for one variable against constants. **Loops**: `for` for a known count, `while` until

something happens, `do-while` to act before checking, for-each to visit every element. A **tracing table** shows exactly what a loop does.

- Profitina Laundry now has `Customer` and `Service`, which price a bag exactly, in integer rupiah, by Equation (3.9) and Equation (3.10).

Chapter 4 works through problems of exactly this kind, in both languages, and Chapter 5 adds arrays, lists, keyboard input, exceptions and files, so that the laundry can finally keep more than a handful of orders.

3.16 Exercises

Solve each exercise in both Java and Python. For the programming exercises, predict the output by hand first, then run your program and compare.

Exercise 3.1 (anatomy). In `Customer` (**Listing 3.53**), list the fields, the constructor and the methods. Which fields cannot change after construction, and how does each language express that? What would go wrong if the constructor said `name = name;` instead of `this.name = name;`?

Exercise 3.2 (references). Predict the output, then draw the object diagram after the last line.

```
Dog a = new Dog("Rex", "Beagle", 2, 10.0);
Dog b = new Dog("Rex", "Beagle", 2, 10.0);
Dog c = a;
c.setAge(3);
b = c;
System.out.println(a.getAge() + " " + b.getAge() + " " + (a == b));
```

Exercise 3.3 (a new class). Write a class `WashingMachine` with an identifier, a capacity in kilograms, a busy flag and the current load. The method `load(kg)` starts a wash and returns `true` only if the machine is free and `0 < kg <= capacity`; otherwise it returns `false` and changes nothing. `unload()` frees the machine. Give it a `toString`, and a `main` that tries loads of 9.5, 6.0 and 2.0 kg on an 8 kg machine.

Exercise 3.4 (strings). Write a program that takes a full name such as `"  siti nur aminah "` and prints the number of words, the initials in capitals (`S.N.A.`), and a bag tag made of the last name with a capital first letter followed by the first initial (`Aminah, S.`). Extra spaces must not matter.

Exercise 3.5 (expressions). Give the value and the type of each Java expression, and the value of its Python counterpart (`/` becomes `//` where Java divides integers): `7 / 2`, `7 / 2.0`, `7 % 3`, `(double) 7 / 2`, `(int) 7.9`, `1000000 * 3000`, `'A' + 2`, `"1" + 2 + 3`, `1 + 2 + "3"`, `-7 / 2`.

Exercise 3.6 (tracing). Trace this loop with a tracing table and state what it computes.

```
int x = 48, y = 18;
while (y != 0) {
    int t = x % y;
    x = y;
    y = t;
}
System.out.println(x);
```

Exercise 3.7 (nested loops). Print a multiplication table for 1 to 5, with a header row and columns 4 characters wide.

Exercise 3.8 (while loop). How many whole years does it take for the GBP 100 of Section 3.11.3 to reach at least GBP 200 at 2.3%? Compare your answer with the rule of thumb known as the *rule of 72*,

$$n \approx \frac{72}{\text{rate}} \tag{3.11}$$

Exercise 3.9 (using the laundry classes). Using `Service` and `Customer` unchanged, print a price table for *Wash & Fold* for 1 to 6 kg, with four columns: guest, guest express, member, member express.

Exercise 3.10 (division and remainder). A customer pays a bill with Indonesian money. Given an amount in rupiah, print how many of each note and coin to use, largest first, with the values 100000, 50000, 20000, 10000, 5000, 2000, 1000, 500, 200, 100 and 50. Test with Rp178,950, the day's total from **Listing 3.55**.

3.17 Solutions to the Exercises

Solution 3.1. Fields: `name`, `phone`, `member`, `visits`. Constructor: `Customer(String name, String phone, boolean member)`, which also sets `visits` to 0. Methods: `getName`, `getPhone`, `isMember`, `setMember`, `getVisits`, `recordVisit`, `maskedPhone`, `toString`. The fields `name` and `phone` never change: Java marks them `final`, so the compiler rejects any second assignment; Python has no such keyword, so the class offers no setter and the underscore asks others not to touch the field. With `name = name;` the parameter would be assigned to itself and the field never assigned. Because the field is `final`, Java then refuses to compile the constructor ("variable name might not have been initialized"); without `final`, the program would compile and silently print `null` as the name.

Solution 3.2. The output is `3 3 true`. After the first two lines there are two separate objects. `c = a` makes `c` an alias of the first dog, so `c.setAge(3)` changes the first dog. `b = c` moves `b`'s arrow to the first dog too, so all three variables point to one object with age 3, and `a == b` is `true`. The second dog, created on line 2, now has no arrow pointing to it: in the object diagram it is crossed out as garbage.

Solution 3.3. The guard in `load` rejects a busy machine, a non-positive load and an overload in one condition joined by `||`, and it returns *before* changing any field, so a refused load leaves the machine untouched. **Listing 3.59** and **Listing 3.60** give the class. The main method is placed inside the class, which is convenient for a small exercise.

Listing 3.59 `WashingMachine.java`: solution to Exercise 3.3.

```
public class WashingMachine {
    private final String id;
    private final double capacityKg;
    private boolean busy;
    private double loadKg;

    public WashingMachine(String id, double capacityKg) {
        this.id = id;
        this.capacityKg = capacityKg;
        this.busy = false;
        this.loadKg = 0.0;
    }

    /** Starts a wash if the machine is free and the load fits. */
    public boolean load(double kg) {
        if (busy || kg <= 0 || kg > capacityKg) {
            return false;
        }
        busy = true;
        loadKg = kg;
        return true;
    }

    public void unload() {
        busy = false;
        loadKg = 0.0;
    }

    public boolean isBusy() {
        return busy;
    }

    @Override
    public String toString() {
        String state = busy ? "busy with " + loadKg + " kg" : "free";
        return "Machine " + id + " (" + capacityKg + " kg): " + state;
    }
```

Listing 3.59 (Continued) *WashingMachine.java: solution to Exercise 3.3.*

```
    public static void main(String[] args) {
        WashingMachine m1 = new WashingMachine("M1", 8.0);
        System.out.println(m1);
        System.out.println("load 9.5 kg -> " + m1.load(9.5));
        System.out.println("load 6.0 kg -> " + m1.load(6.0));
        System.out.println("load 2.0 kg -> " + m1.load(2.0));
        System.out.println(m1);
        m1.unload();
        System.out.println(m1);
    }
}
```

```
Machine M1 (8.0 kg): free
load 9.5 kg -> false
load 6.0 kg -> true
load 2.0 kg -> false
Machine M1 (8.0 kg): busy with 6.0 kg
Machine M1 (8.0 kg): free
```

Listing 3.60 `washing_machine.py`: solution to Exercise 3.3 in Python.

```python
class WashingMachine:
    def __init__(self, machine_id, capacity_kg):
        self._id = machine_id
        self._capacity_kg = capacity_kg
        self._busy = False
        self._load_kg = 0.0

    def load(self, kg):
        """Starts a wash if the machine is free and the load fits."""
        if self._busy or kg <= 0 or kg > self._capacity_kg:
            return False
        self._busy = True
        self._load_kg = kg
        return True

    def unload(self):
        self._busy = False
        self._load_kg = 0.0

    def is_busy(self):
        return self._busy

    def __str__(self):
        state = "busy with " + str(self._load_kg) + " kg" if self._busy else "free"
        return "Machine " + self._id + " (" + str(self._capacity_kg) + " kg): " + state


def main():
    m1 = WashingMachine("M1", 8.0)
    print(m1)
    print("load 9.5 kg -> " + str(m1.load(9.5)))
    print("load 6.0 kg -> " + str(m1.load(6.0)))
    print("load 2.0 kg -> " + str(m1.load(2.0)))
    print(m1)
    m1.unload()
    print(m1)


if __name__ == "__main__":
    main()
```

```
Machine M1 (8.0 kg): free
load 9.5 kg -> False
load 6.0 kg -> True
load 2.0 kg -> False
Machine M1 (8.0 kg): busy with 6.0 kg
Machine M1 (8.0 kg): free
```

The first load is refused because 9.5 > 8.0; the second is accepted; the third is refused because the machine is busy.

Solution 3.4. The key is to split on *one or more* spaces after trimming: Java's `split(" +")` uses the pattern `" +"`, meaning "one or more spaces", and Python's `split()` without an argument does the same automatically. Each initial is `charAt(0)` (Python `part[0]`) converted to upper case, as in **Listing 3.61** and **Listing 3.62**.

Listing 3.61 `Initials.java`: solution to Exercise 3.4.

```java
public class Initials {

    public static void main(String[] args) {
        String fullName = args.length > 0 ? args[0] : "  siti nur aminah ";
        String[] parts = fullName.trim().split(" +");   // one or more spaces
        String initials = "";
        for (String part : parts) {
            initials = initials + Character.toUpperCase(part.charAt(0)) + ".";
        }
        System.out.println("Name     : [" + fullName.trim() + "]");
        System.out.println("Words    : " + parts.length);
        System.out.println("Initials : " + initials);
        String last = parts[parts.length - 1];
        String tag = Character.toUpperCase(last.charAt(0)) +
last.substring(1).toLowerCase();
        System.out.println("Bag tag  : " + tag + ", " + initials.substring(0, 2));
    }
}
```

```
Name     : [siti nur aminah]
Words    : 3
Initials : S.N.A.
Bag tag  : Aminah, S.
```

Listing 3.62 `initials.py`: solution to Exercise 3.4 in Python.

```python
import sys


def main():
    full_name = sys.argv[1] if len(sys.argv) > 1 else "  siti nur aminah "
    parts = full_name.split()  # split() with no argument skips extra spaces
    initials = ""
    for part in parts:
        initials = initials + part[0].upper() + "."
    print("Name     : [" + full_name.strip() + "]")
    print("Words    : " + str(len(parts)))
    print("Initials : " + initials)
    last = parts[-1]
    tag = last[0].upper() + last[1:].lower()
    print("Bag tag  : " + tag + ", " + initials[0:2])


if __name__ == "__main__":
    main()
```

```
Name     : [siti nur aminah]
Words    : 3
Initials : S.N.A.
Bag tag  : Aminah, S.
```

Solution 3.5. **Table 3.15** gives the answers; each was checked by running it.

Table 3.15 Values of the expressions in Exercise 3.5.

Java expression	Java value (type)	Python counterpart	Python value
`7 / 2`	`3` (`int`)	`7 // 2`	`3`
`7 / 2.0`	`3.5` (`double`)	`7 / 2.0`	`3.5`
`7 % 3`	`1` (`int`)	`7 % 3`	`1`
`(double) 7 / 2`	`3.5` (`double`)	`7 / 2`	`3.5`
`(int) 7.9`	`7` (`int`)	`int(7.9)`	`7`
`1000000 * 3000`	`-1294967296` (`int`, overflow)	`1000000 * 3000`	`3000000000`
`'A' + 2`	`67` (`int`)	`chr(ord('A') + 2)`	`'C'`
`"1" + 2 + 3`	`"123"` (`String`)	`"1" + str(2) + str(3)`	`'123'`
`1 + 2 + "3"`	`"33"` (`String`)	`str(1 + 2) + "3"`	`'33'`
`-7 / 2`	`-3` (`int`)	`-7 // 2`	`-4`

Two rows deserve a word. In `"1" + 2 + 3`, Java works from left to right: `"1" + 2` is the string `"12"`, and `"12" + 3` is `"123"`. In `1 + 2 + "3"` the first `+` adds two numbers, giving 3, and only then meets a string, so the result is `"33"`. And `1000000 * 3000` = 3×10^9 exceeds $2^{31} - 1$, so by Equation (3.2) it wraps to $3 \times 10^9 - 2^{32} = -1\,294\,967\,296$.

Solution 3.6. **Table 3.16** traces the loop.

Table 3.16 Tracing the loop of Exercise 3.6 for $x = 48$, $y = 18$.

Iteration	x	y	t = x % y	New x	New y
1	48	18	12	18	12
2	18	12	6	12	6
3	12	6	0	6	0

Now `y` is 0, the loop ends, and the program prints 6. This is Euclid's algorithm for the **greatest common divisor**: 6 is the largest number that divides both 48 and 18. It works because any number that divides both `x` and `y` also divides `x % y`, so each step replaces the pair by a smaller pair with the same common divisors.

Solution 3.7. An outer loop runs over the rows, and for each row an inner loop runs over the columns; the inner loop completes all its iterations for each single iteration of the outer loop. Loops placed inside loops like this are called **nested loops**; see **Listing 3.63** and **Listing 3.64**.

Listing 3.63 `MultiplicationTable.java`: solution to Exercise 3.7.

```java
public class MultiplicationTable {

    public static void main(String[] args) {
        int n = 5;
        System.out.print("   |");
        for (int col = 1; col <= n; col++) {
            System.out.printf("%4d", col);
        }
        System.out.println();
        System.out.println("---+" + "-".repeat(4 * n));
        for (int row = 1; row <= n; row++) {
            System.out.printf("%2d |", row);
            for (int col = 1; col <= n; col++) {
                System.out.printf("%4d", row * col);
            }
            System.out.println();
        }
    }
}
```

```
   |   1   2   3   4   5
---+--------------------
 1 |   1   2   3   4   5
 2 |   2   4   6   8  10
 3 |   3   6   9  12  15
 4 |   4   8  12  16  20
 5 |   5  10  15  20  25
```

Listing 3.64 `multiplication_table.py`: solution to Exercise 3.7 in Python.

```python
def main():
    n = 5
    print("   |", end="")
    for col in range(1, n + 1):
        print(f"{col:4d}", end="")
    print()
    print("---+" + "-" * (4 * n))
    for row in range(1, n + 1):
        print(f"{row:2d} |", end="")
        for col in range(1, n + 1):
            print(f"{row * col:4d}", end="")
        print()


if __name__ == "__main__":
    main()
```

```
   |   1   2   3   4   5
---+--------------------
 1 |   1   2   3   4   5
 2 |   2   4   6   8  10
 3 |   3   6   9  12  15
 4 |   4   8  12  16  20
 5 |   5  10  15  20  25
```

Solution 3.8. We do not know in advance how many years are needed; we loop *until* the balance reaches 200. That is a `while` loop, in **Listing 3.65** and **Listing 3.66**.

Listing 3.65 `DoublingYears.java`: solution to Exercise 3.8.

```java
public class DoublingYears {

    public static void main(String[] args) {
        double capital = 100;
        double interestRate = 2.3;
        double balance = capital;
        int years = 0;
        while (balance < 2 * capital) {
            balance = balance * (1 + 0.01 * interestRate);
            years++;
        }
        System.out.printf("Doubled after %d years: GBP %.2f%n", years, balance);
        System.out.printf("Rule of 72 estimate: %.1f years%n", 72 / interestRate);
    }
}
```

```
Doubled after 31 years: GBP 202.37
Rule of 72 estimate: 31.3 years
```

Listing 3.66 `doubling_years.py`: solution to Exercise 3.8 in Python.

```python
def main():
    capital = 100.0
    interest_rate = 2.3
    balance = capital
    years = 0
    while balance < 2 * capital:
        balance = balance * (1 + 0.01 * interest_rate)
        years += 1
    print(f"Doubled after {years} years: GBP {balance:.2f}")
    print(f"Rule of 72 estimate: {72 / interest_rate:.1f} years")


if __name__ == "__main__":
    main()
```

```
Doubled after 31 years: GBP 202.37
Rule of 72 estimate: 31.3 years
```

It takes 31 years; after 30 years the balance is still just under 200 (about 197.8), which is why the loop runs once more. Equation (3.11) estimates $72/2.3 \approx 31.3$ years, an excellent approximation. The same answer can be read off **Figure 3.18**, where the line climbs by a factor of two roughly every 31 years.

Solution 3.9. The table reuses the classes without changing them, which is the point of writing classes: the price rules live in one place, `Service.price`. The solution is **Listing 3.67** and **Listing 3.68**.

Listing 3.67 `PriceTable.java`: solution to Exercise 3.9.

```java
public class PriceTable {

    public static void main(String[] args) {
        Service washFold = new Service("WF", "Wash & Fold", 7000, 3.0);
        Customer guest = new Customer("Walk-in", "0000", false);
        Customer member = new Customer("Member", "0000", true);

        System.out.println(" kg   guest   guest+exp   member   member+exp");
        for (double kg = 1.0; kg <= 6.0; kg += 1.0) {
            System.out.printf("%3.0f %7d %11d %8d %12d%n", kg,
                    washFold.price(kg, false, guest),
                    washFold.price(kg, true, guest),
                    washFold.price(kg, false, member),
                    washFold.price(kg, true, member));
        }
    }
}
```

```
 kg   guest   guest+exp   member   member+exp
  1   21000       31500    18900       28350
  2   21000       31500    18900       28350
  3   21000       31500    18900       28350
  4   28000       42000    25200       37800
  5   35000       52500    31500       47250
  6   42000       63000    37800       56700
```

Listing 3.68 `price_table.py`: solution to Exercise 3.9 in Python.

```python
from customer import Customer
from service import Service


def main():
    wash_fold = Service("WF", "Wash & Fold", 7000, 3.0)
    guest = Customer("Walk-in", "0000", False)
    member = Customer("Member", "0000", True)

    print(" kg   guest   guest+exp   member   member+exp")
    for kg in range(1, 7):
        print(f"{kg:3d} {wash_fold.price(kg, False, guest):7d} "
              f"{wash_fold.price(kg, True, guest):11d} "
              f"{wash_fold.price(kg, False, member):8d} "
              f"{wash_fold.price(kg, True, member):12d}")


if __name__ == "__main__":
    main()
```

```
 kg   guest   guest+exp   member   member+exp
  1   21000       31500    18900       28350
  2   21000       31500    18900       28350
  3   21000       31500    18900       28350
  4   28000       42000    25200       37800
  5   35000       52500    31500       47250
  6   42000       63000    37800       56700
```

The first three rows are equal because of the 3 kg minimum in Equation (3.9). The Java loop counts with a `double`, `kg += 1.0`; this is safe here because whole numbers are represented exactly in a `double`, but counting in steps of 0.1 would run into the error of **Figure 3.14**, and the Python version counts with integers instead.

Solution 3.10. For each value v, largest first, the number of notes is the integer quotient `rest / v` and what remains is `rest % v`, which is Equation (3.4) used twice per step; see **Listing 3.69** and **Listing 3.70**.

Listing 3.69 `BanknoteBreakdown.java`: solution to Exercise 3.10.

```java
public class BanknoteBreakdown {

    public static void main(String[] args) {
        long amount = args.length > 0 ? Long.parseLong(args[0]) : 178950;
        long[] values = {100000, 50000, 20000, 10000, 5000, 2000, 1000, 500, 200, 100, 50};

        System.out.printf("Rp%,d is paid as:%n", amount);
        long rest = amount;
        for (long v : values) {
            long count = rest / v;     // how many of this note fit
            rest = rest % v;           // what is left over
            if (count > 0) {
                System.out.printf("  %d x Rp%,d%n", count, v);
            }
        }
        if (rest > 0) {
            System.out.println("  cannot pay the last Rp" + rest);
        }
    }
}
```

```
Rp178,950 is paid as:
  1 x Rp100,000
  1 x Rp50,000
  1 x Rp20,000
  1 x Rp5,000
  1 x Rp2,000
  1 x Rp1,000
  1 x Rp500
  2 x Rp200
  1 x Rp50
```

Listing 3.70 `banknote_breakdown.py`: solution to Exercise 3.10 in Python.

```python
import sys


def main():
    amount = int(sys.argv[1]) if len(sys.argv) > 1 else 178950
    values = [100000, 50000, 20000, 10000, 5000, 2000, 1000, 500, 200, 100, 50]

    print(f"Rp{amount:,d} is paid as:")
    rest = amount
    for v in values:
        count = rest // v  # how many of this note fit
        rest = rest % v    # what is left over
        if count > 0:
            print(f"  {count} x Rp{v:,d}")
    if rest > 0:
        print("  cannot pay the last Rp" + str(rest))


if __name__ == "__main__":
    main()
```

```
Rp178,950 is paid as:
  1 x Rp100,000
  1 x Rp50,000
  1 x Rp20,000
  1 x Rp5,000
  1 x Rp2,000
  1 x Rp1,000
  1 x Rp500
  2 x Rp200
  1 x Rp50
```

Check: $100\,000 + 50\,000 + 20\,000 + 5\,000 + 2\,000 + 1\,000 + 500 + 2 \times 200 + 50 = 178\,950$. The method of always taking the largest value that fits is called a **greedy** method; for Indonesian money it always gives the fewest pieces, but for some other sets of values it does not, a question studied in courses on algorithms.

References

[1] G. van Rossum, B. Warsaw, and A. Coghlan, "PEP 8 – Style Guide for Python Code," Python Software Foundation, 2001 (updated 2013). [Online]. Available: https://peps.python.org/pep-0008/

[2] D. J. Barnes and M. Kölling, *Objects First with Java: A Practical Introduction Using BlueJ*, 6th ed. Harlow, UK: Pearson, 2016.

[3] The JUnit Team, *JUnit User Guide*, version 6.1.3, Aug. 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://docs.junit.org/current/

[4] Oracle, "Class String," *Java SE 21 & JDK 21 API Specification*, 2023. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html

[5] Python Software Foundation, "Built-in Types: Text Sequence Type — str," *The Python Standard Library*, 2025. [Online]. Available: https://docs.python.org/3/library/stdtypes.html

[6] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*. Redwood City, CA, USA: Oracle, 2023.

[7] *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019, 2019.

[8] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, 1991.

[9] G. Bierman, "JEP 361: Switch Expressions," OpenJDK, 2019. [Online]. Available: https://openjdk.org/jeps/361

[10] D. F. Moisset, "PEP 636 – Structural Pattern Matching: Tutorial," Python Software Foundation, 2020. [Online]. Available: https://peps.python.org/pep-0636/

[11] R. Sedgewick and K. Wayne, *Computer Science: An Interdisciplinary Approach*. Boston, MA, USA: Addison-Wesley, 2016.

[12] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

Chapter 4

Evaluation and Discussion I: From a Plain Specification to a Tested Class

This chapter adds no new language features. Instead, it takes everything Chapters 1 to 3 have given you — the workbench, classes and objects, strings, types, conditionals and loops — and puts it to work on fourteen complete problems. Seven of them are the classic first problems of an object-oriented programming course: adding fractions, measuring the angle between the hands of a clock, a computer account, complex numbers, a weight in pounds and kilograms, a fraction class, and a small fight between a player and an enemy. Seven more are new and come from everyday life: rupiah amounts, times of day, a Profitina Laundry receipt, a tidy customer name, password rules, loops traced by hand, and leap years. Every problem is solved from nothing, very slowly, in the same order every time: the problem restated in plain words, two or three examples worked by hand, the classes, fields and methods identified, a design drawn, a Java solution and its test driver, a Python solution with `unittest`, the real output of both, a line-by-line explanation, a *tempting wrong solution* with the exact way it fails, and a *check yourself* variant with its answer. By the end of the chapter you will have a method for attacking any small programming problem, and fourteen fully worked examples of that method to lean on.

4.1 Where We Left Off

Before we solve anything, let us lay out the toolkit. Every problem in this chapter reaches for a handful of these tools, and naming the tool you need is already half of the work.

From **Chapter 1** you have a workbench: a Java Development Kit (JDK), a Python interpreter, and Visual Studio Code with its Java and Python extensions. You know how to create a file, run it, and read what it prints. You also have the central idea of the book: an *object* bundles some data together with the operations that are allowed on that data.

From **Chapter 2** you can edit, run and *debug*: set a breakpoint, step through a program one line at a time, and watch the value of a variable change. Several of the tempting wrong solutions in this chapter are exactly the kind of bug a debugger finds in a minute, and we will say so each time.

From **Chapter 3** you have the language itself. A *class* is a blueprint; an *object* is one thing built from it with `new` (Java) or by calling the class (Python). A class has *fields* (the data each object carries), a *constructor* (the code that fills the fields when an object is born) and *methods* (the operations). You know the primitive types `int`, `long` and `double` and the traps hidden in them, above all integer division; you know `String` and its methods `length`, `charAt`, `substring` and `equals`; you know `if`, `else if`, `else` and `switch`; and you know the `while` and `for` loops. Chapter 3 ended with small exercises such as the area of a circle, a converter from imperial masses to kilograms, and compound interest; Problem 4.5 grows directly out of that converter.

Table 4.1 shows which tool each problem of this chapter exercises most. Use it as a map: if you feel unsure about loops, start with Problems 4.11 to 4.13; if classes still feel strange, start with Problems 4.3 and 4.5.

Table 4.1 The fourteen problems of this chapter and the tools from Chapters 1 to 3 that each one exercises most.

Problem	Topic	Main tools from Chapters 1–3
4.1	Fraction calculator	`int` arithmetic, static methods, string concatenation
4.2	Analog clock angle	`double`, the `%` operator, rounding, normalization
4.3	Computer account	private fields, constructor, `equals` on strings
4.4	Complex numbers	methods that return new objects, `toString`, `if`
4.5	Weight conversion	constants, a derived value computed on demand
4.6	Fraction class	objects as parameters and results, immutability
4.7	Player and enemy	two collaborating classes, `if`, a `while` loop
4.8	Money in rupiah	`long`, a loop over the characters of a string
4.9	Time of day	one field instead of two, wrap-around with `%`, formatting
4.10	Laundry receipt	`switch`, rounding up, one object inside another
4.11	Tidying a name	`charAt`, character tests, building a string in a loop
4.12	Password rules	counting in a loop, deciding after the loop
4.13	Tracing loops	trace tables, `while`, Euclid's algorithm
4.14	Leap years	Boolean expressions, `switch`, a method that returns the next object

4.2 How to Approach a Programming Problem

A beginner who is handed a problem usually opens the editor and starts typing `public class`. An experienced programmer usually does not touch the keyboard for the first few minutes. The difference is not talent; it is a routine. The mathematician George Pólya described such a routine for mathematical problems — understand the problem, devise a plan, carry it out, look back [1] — and the same four movements, stretched into eight small steps, serve programmers just as well. **Figure 4.1** shows the eight steps we will follow for every problem in this chapter.

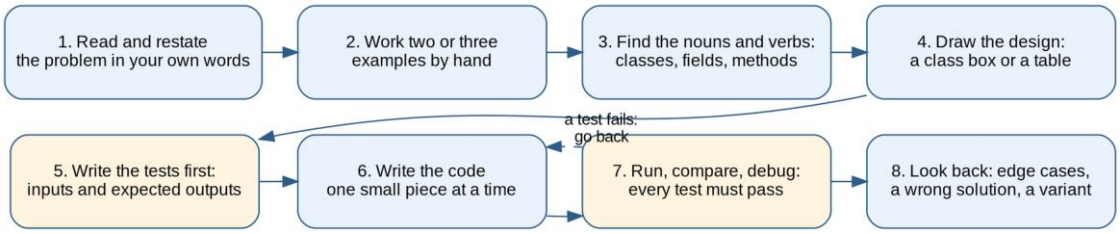


Figure 4.1 The eight steps used for every problem in this chapter; the dashed arrow is the loop between writing code and running the tests.

4.2.1 The eight steps

Step 1 — Read the problem and restate it in your own words. Read the statement twice. Then write, in one or two sentences of your own, what goes *in* and what must come *out*. If you cannot do this, you do not yet understand the problem, and no amount of code will help. Circle every number in the statement: numbers are usually the examples you will test with.

Step 2 — Work two or three examples by hand. Take the examples the statement gives you, and compute the answers with pencil and paper *before* writing any code. Write down every intermediate number. This is where you discover whether you really understand the formula, and

it gives you the expected outputs for step 5. Add one example of your own at an edge: zero, a negative number, the largest allowed value, an empty string.

Step 3 — Find the nouns and the verbs. In an object-oriented program, the *nouns* of the statement ("a fraction", "an account", "a receipt") are candidates for classes; the *properties* of each noun ("its numerator", "its password", "its total") are candidates for fields; and the *verbs* ("add", "change the password", "print") are candidates for methods. Make a list.

Step 4 — Draw the design. Draw each class as a box with three compartments: its name, its fields with their types, and its methods with their parameters and results. This is a simplified form of a *UML class diagram* (UML is the Unified Modeling Language, the standard notation for drawing software designs). A minus sign marks something private, a plus sign something public. We draw these boxes with ordinary characters in this chapter, so you can copy them onto paper.

Step 5 — Write the tests first. Turn the examples of step 2 into checks: "for this input, the method must return exactly this". Writing them before the code sounds backwards, but it forces you to decide precisely what "correct" means while your head is still clear.

Step 6 — Write the code, one small piece at a time. Write the fields and the constructor first, compile, and run. Then one method, compile, and run. A program that grows in small, tested steps never presents you with fifty errors at once.

Step 7 — Run, compare, and debug. Run the tests. When a check fails, do not guess: set a breakpoint in the failing method (Chapter 2) and watch the variables until you see the first value that differs from your hand calculation. That difference *is* the bug. Go back to step 6, fix it, and run all the tests again — the dashed arrow in **Figure 4.1**.

Step 8 — Look back. When everything passes, ask three questions. Which input did I not test? What is the most *tempting* wrong way to solve this, and why does it fail? Could I solve a slightly different version? These three questions are exactly the last three parts of every problem in this chapter.

> **WHY EXAMPLES BY HAND COME FIRST.** A program can only be as right as your understanding of the problem. If you cannot compute $\frac{1}{2}+\frac{1}{3}$ on paper, you cannot check a program that claims it is $\frac{5}{6}$. Every expected value in the test drivers of this chapter was first computed by hand in the "Understanding it by hand" part of its problem.

4.2.2 How the tests in this chapter are written

Professional Java programs are tested with JUnit, today JUnit 6 (its test API, called *Jupiter*, is the one JUnit 5 introduced, so tests written for JUnit 5 run unchanged), a library in which each check is a method marked `@Test` and a statement such as `assertEquals(5, result)` compares the expected with the actual value [2]. JUnit is a separate download, and this book does not want your first tests to depend on getting a library onto your class path. So every Java solution in this chapter comes with a *plain test driver*: an ordinary class with a `main` method and a small helper method `check(label, expected, actual)` that prints `PASS` or `FAIL` and counts the results. The helper compares with `equals`, which is exactly what `assertEquals` does inside JUnit, so moving a driver to JUnit later is mechanical: each `check(label, expected, actual)` becomes `assertEquals(expected, actual, label)` inside a method marked `@Test`. The README of the chapter's code folder describes that step in VS Code: open the Testing view, choose *Enable Java Tests*, pick *JUnit Jupiter*, and VS Code downloads the library and shows a green or red mark beside every test.

Python ships with a testing library in its standard library, `unittest` [3]. Each Python solution in this chapter therefore contains a class that extends `unittest.TestCase`, whose methods named `test_…` call `self.assertEqual(actual, expected)`, `self.assertTrue(…)` or `self.assertAlmostEqual(…)`. The last lines of each file run a short demonstration and then the tests, so that one command — `python3 file.py` — shows both. The same tests also run with `python3 -m unittest file` or with the Testing view of VS Code.

One practical note about the quoted output: `unittest` reports how long the tests took, such as `Ran 4 tests in 0.000s`. Your computer may print `0.001s`; that line is the only one that may differ from the book.

4.2.3 How each problem is laid out

Every problem section has the same parts, so that you always know where you are:

1. **The problem** — the statement, in a box.
2. **Understanding it by hand** — the statement restated in plain words and the examples computed on paper (steps 1 and 2).
3. **Classes, fields and methods** — the nouns and verbs, and a class box (steps 3 and 4).
4. **The Java solution** — the class, its test driver and their real output (steps 5 to 7).
5. **The Python solution** — the same design in Python, with `unittest`, and its real output.
6. **Line by line** — what every important line does and why.
7. **A tempting wrong solution** — a program that looks right, its real output, and why it fails (step 8).
8. **Check yourself** — a variant of the problem with its answer.

All programs are in the chapter's code folder, `Chapter04/java` and `Chapter04/python`; the tempting wrong solutions in Java live in `Chapter04/java/tempting`, because one of them is *meant* not to compile and would otherwise stop `javac *.java` from compiling the rest. The Java solutions keep the class names of the classic problem set, such as `Q11Fraction` and `Q16Fraction`: the prefix keeps the two fraction classes of Problems 4.1 and 4.6 apart in one folder.

4.3 Problem 4.1 — Fraction Calculator

4.3.1 The problem

> **PROBLEM 4.1.** Let `e1` and `d1` represent the fraction $\frac{e_1}{d_1}$, and correspondingly `e2` and `d2` the fraction $\frac{e_2}{d_2}$, where e_1 and e_2 are integers and d_1 and d_2 are positive integers. Write a Java class `Q11Fraction` that computes the numbers `es`, `ds` and `ep`, `dp`, which stand for the sum and the product of the two fractions. For example, $\frac{1}{2}+\frac{1}{3}=\frac{5}{6}$ and $\frac{1}{2}\cdot\frac{1}{3}=\frac{1}{6}$. Addition and multiplication of fractions are defined by Equation (4.1) and Equation (4.2). Test the class for $\frac{1}{2}+\frac{1}{3}$, $\frac{1}{3}+\frac{3}{4}$, $\frac{1}{2}\cdot\frac{2}{3}$ and $\frac{1}{4}\cdot\frac{2}{3}$.

$$\frac{e_1}{d_1}+\frac{e_2}{d_2}=\frac{e_1 d_2+e_2 d_1}{d_1 d_2} \tag{4.1}$$

$$\frac{e_1}{d_1}\cdot\frac{e_2}{d_2}=\frac{e_1 e_2}{d_1 d_2} \tag{4.2}$$

4.3.2 Understanding it by hand

In plain words: we are given four whole numbers, which describe two fractions, and we must produce four more whole numbers, which describe their sum and their product. The letter e stands for the top of a fraction (the *numerator*) and d for the bottom (the *denominator*). The answer is *not* a decimal number such as 0.8333; it is a pair of whole numbers.

Why is Equation (4.1) true? To add two fractions we must first cut both into pieces of the same size. If we multiply the top and bottom of $\frac{e_1}{d_1}$ by d_2, and the top and bottom of $\frac{e_2}{d_2}$ by d_1, both fractions have the common bottom $d_1 d_2$, and then we simply add the tops. Multiplication needs no common bottom: "a half of a third" is one piece out of $2 \cdot 3 = 6$, so tops multiply and bottoms multiply.

Now the four examples, one step per line:

- $\frac{1}{2} + \frac{1}{3}$: $e_s = 1 \cdot 3 + 1 \cdot 2 = 3 + 2 = 5$ and $d_s = 2 \cdot 3 = 6$, so the sum is $\frac{5}{6}$.
- $\frac{1}{3} + \frac{3}{4}$: $e_s = 1 \cdot 4 + 3 \cdot 3 = 4 + 9 = 13$ and $d_s = 3 \cdot 4 = 12$, so the sum is $\frac{13}{12}$.
- $\frac{1}{2} \cdot \frac{2}{3}$: $e_p = 1 \cdot 2 = 2$ and $d_p = 2 \cdot 3 = 6$, so the product is $\frac{2}{6}$.
- $\frac{1}{4} \cdot \frac{2}{3}$: $e_p = 1 \cdot 2 = 2$ and $d_p = 4 \cdot 3 = 12$, so the product is $\frac{2}{12}$.

Notice that $\frac{2}{6}$ equals $\frac{1}{3}$ and $\frac{2}{12}$ equals $\frac{1}{6}$. The problem does not ask us to reduce, so the program should print $\frac{2}{6}$, and our tests will expect exactly that. (Reducing is the *check yourself* of Problem 4.6.) Our own edge example: a negative first fraction, $-\frac{1}{2} + \frac{1}{3} = \frac{-3+2}{6} = \frac{-1}{6}$.

4.3.3 Classes, fields and methods

There is no object with a memory here yet: the problem only asks for four computations. So the class `Q11Fraction` has no fields, only *static* methods — methods that belong to the class itself and are called as `Q11Fraction.sumNumerator(…)` without creating an object (Chapter 3). One method per number we must compute keeps each formula in exactly one place:

```
+-------------------------------------------------------------+
| Q11Fraction                                                 |
+-------------------------------------------------------------+
| (no fields)                                                 |
+-------------------------------------------------------------+
| + sumNumerator(e1, d1, e2, d2) : int        (static)        |
| + sumDenominator(d1, d2) : int              (static)        |
| + productNumerator(e1, e2) : int            (static)        |
| + productDenominator(d1, d2) : int          (static)        |
| + printSum(e1, d1, e2, d2), printProduct(e1, d1, e2, d2)    |
| + main(args)                                                |
+-------------------------------------------------------------+
```

That a fraction has to be passed around as two loose numbers is clumsy; hold on to that feeling, because Problem 4.6 cures it.

4.3.4 The Java solution

Listing 4.1 is the class. Its `main` method prints the four results the problem asks for.

Listing 4.1 `Q11Fraction.java`: the sum and product of two fractions with whole numbers only.

```
/**
 * Problem 4.1: the sum and the product of two fractions e1/d1 and e2/d2,
 * using whole numbers only:
 *   e1/d1 + e2/d2 = (e1*d2 + e2*d1) / (d1*d2)
 *   e1/d1 * e2/d2 = (e1*e2) / (d1*d2)
 * The results are not reduced to lowest terms.
 */
public class Q11Fraction {

    public static int sumNumerator(int e1, int d1, int e2, int d2) {
        return e1 * d2 + e2 * d1;
    }

    public static int sumDenominator(int d1, int d2) {
        return d1 * d2;
    }

    public static int productNumerator(int e1, int e2) {
        return e1 * e2;
    }

    public static int productDenominator(int d1, int d2) {
        return d1 * d2;
    }

    public static void printSum(int e1, int d1, int e2, int d2) {
        int es = sumNumerator(e1, d1, e2, d2);
        int ds = sumDenominator(d1, d2);
        System.out.println(e1 + "/" + d1 + " + " + e2 + "/" + d2 + " = " + es + "/" + ds);
    }

    public static void printProduct(int e1, int d1, int e2, int d2) {
        int ep = productNumerator(e1, e2);
        int dp = productDenominator(d1, d2);
        System.out.println(e1 + "/" + d1 + " * " + e2 + "/" + d2 + " = " + ep + "/" + dp);
    }

    public static void main(String[] args) {
        printSum(1, 2, 1, 3);
        printSum(1, 3, 3, 4);
        printProduct(1, 2, 2, 3);
        printProduct(1, 4, 2, 3);
    }
}
```

Compiled with `javac Q11Fraction.java` and run with `java Q11Fraction`, it prints:

```
1/2 + 1/3 = 5/6
1/3 + 3/4 = 13/12
1/2 * 2/3 = 2/6
1/4 * 2/3 = 2/12
```

The four lines agree with our hand computation. The test driver in **Listing 4.2** turns the same examples, plus the negative one, into automatic checks.

Listing 4.2 `Q11FractionTest.java`: a plain test driver for Problem 4.1.

```java
/** Plain test driver for Q11Fraction: run it with "java Q11FractionTest". */
public class Q11FractionTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        check("1/2 + 1/3 numerator", 5, Q11Fraction.sumNumerator(1, 2, 1, 3));
        check("1/2 + 1/3 denominator", 6, Q11Fraction.sumDenominator(2, 3));
        check("1/3 + 3/4 numerator", 13, Q11Fraction.sumNumerator(1, 3, 3, 4));
        check("1/3 + 3/4 denominator", 12, Q11Fraction.sumDenominator(3, 4));
        check("1/2 * 2/3 numerator", 2, Q11Fraction.productNumerator(1, 2));
        check("1/4 * 2/3 denominator", 12, Q11Fraction.productDenominator(4, 3));
        check("-1/2 + 1/3 numerator", -1, Q11Fraction.sumNumerator(-1, 2, 1, 3));
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

```
PASS 1/2 + 1/3 numerator
PASS 1/2 + 1/3 denominator
PASS 1/3 + 3/4 numerator
PASS 1/3 + 3/4 denominator
PASS 1/2 * 2/3 numerator
PASS 1/4 * 2/3 denominator
PASS -1/2 + 1/3 numerator
7 passed, 0 failed
```

4.3.5 The Python solution

Listing 4.3 is the same design in Python: four functions, two printing helpers, and a `unittest` test class.

Listing 4.3 `q11_fraction.py`: Problem 4.1 in Python, with its `unittest` tests.

```
"""Problem 4.1: the sum and the product of two fractions e1/d1 and e2/d2,
using whole numbers only:
    e1/d1 + e2/d2 = (e1*d2 + e2*d1) / (d1*d2)
    e1/d1 * e2/d2 = (e1*e2) / (d1*d2)
The results are not reduced to lowest terms.
"""
import unittest


def sum_numerator(e1, d1, e2, d2):
    return e1 * d2 + e2 * d1


def sum_denominator(d1, d2):
    return d1 * d2


def product_numerator(e1, e2):
    return e1 * e2


def product_denominator(d1, d2):
    return d1 * d2


def print_sum(e1, d1, e2, d2):
    es = sum_numerator(e1, d1, e2, d2)
    ds = sum_denominator(d1, d2)
    print(f"{e1}/{d1} + {e2}/{d2} = {es}/{ds}")


def print_product(e1, d1, e2, d2):
    ep = product_numerator(e1, e2)
    dp = product_denominator(d1, d2)
    print(f"{e1}/{d1} * {e2}/{d2} = {ep}/{dp}")


class TestQ11Fraction(unittest.TestCase):
    def test_sum_half_and_third(self):
        self.assertEqual(sum_numerator(1, 2, 1, 3), 5)
        self.assertEqual(sum_denominator(2, 3), 6)

    def test_sum_third_and_three_quarters(self):
        self.assertEqual(sum_numerator(1, 3, 3, 4), 13)
        self.assertEqual(sum_denominator(3, 4), 12)
```

Listing 4.3 (Continued) *q11_fraction.py: Problem 4.1 in Python, with its unittest tests.*

```
    def test_products(self):
        self.assertEqual(product_numerator(1, 2), 2)
        self.assertEqual(product_denominator(4, 3), 12)

    def test_negative_numerator(self):
        self.assertEqual(sum_numerator(-1, 2, 1, 3), -1)


if __name__ == "__main__":
    print_sum(1, 2, 1, 3)
    print_sum(1, 3, 3, 4)
    print_product(1, 2, 2, 3)
    print_product(1, 4, 2, 3)
    unittest.main(verbosity=2)
```

Running `python3 q11_fraction.py` prints the four results and then the verdict of each test:

```
1/2 + 1/3 = 5/6
1/3 + 3/4 = 13/12
1/2 * 2/3 = 2/6
1/4 * 2/3 = 2/12
test_negative_numerator (__main__.TestQ11Fraction.test_negative_numerator) ... ok
test_products (__main__.TestQ11Fraction.test_products) ... ok
test_sum_half_and_third (__main__.TestQ11Fraction.test_sum_half_and_third) ... ok
test_sum_third_and_three_quarters
(__main__.TestQ11Fraction.test_sum_third_and_three_quarters) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

4.3.6 Line by line

- `public static int sumNumerator(int e1, int d1, int e2, int d2)` — `public` means any class may call it, `static` that it belongs to the class rather than to an object, and `int` that it returns a whole number. The body is Equation (4.1) typed almost letter for letter: `e1 * d2 + e2 * d1`. Java, like school arithmetic, multiplies before it adds, so no parentheses are needed.
- `sumDenominator` and `productDenominator` contain the same expression `d1 * d2`. That is not duplication of *knowledge*: one is "the bottom of a sum" and the other "the bottom of a product", which only happen to agree.
- `System.out.println(e1 + "/" + d1 + " + " + …)` — the `+` operator between a string and a number *concatenates* (joins text), so `1 + "/" + 2` becomes the text `1/2`. This works because Java reads the expression from left to right and the first operand next to a string turns the whole thing into text.
- In the test driver, `check("1/2 + 1/3 numerator", 5, Q11Fraction.sumNumerator(1, 2, 1, 3))` passes the number `5` where the parameter type is `Object`. Java wraps the `int` automatically in an `Integer` object (this is called *autoboxing*), and `expected.equals(actual)` then compares two `Integer` objects by value.
- In Python, `f"{e1}/{d1} + {e2}/{d2} = {es}/{ds}"` is an *f-string*: every expression in braces is replaced by its value. The `if __name__ == "__main__":` block runs only when the file is run directly, not when another file imports it; `unittest.main(verbosity=2)` finds every method whose name begins with `test` and prints one line per test.

4.3.7 A tempting wrong solution

Three ideas come to almost every beginner. **Listing 4.4** shows all three.

Listing 4.4 `WrongFractionSum.java`: three tempting ways to add $\frac{1}{2}$ and $\frac{1}{3}$.

```
/** Three tempting but wrong ways to add 1/2 and 1/3. */
public class WrongFractionSum {
    public static void main(String[] args) {
        int e1 = 1, d1 = 2, e2 = 1, d2 = 3;
        System.out.println("tops and bottoms added: " + (e1 + e2) + "/" + (d1 + d2));
        System.out.println("int division:           " + (e1 / d1 + e2 / d2));
        System.out.println("double division:        " + ((double) e1 / d1 + (double) e2 /
d2));
    }
}
```

```
tops and bottoms added: 2/5
int division:           0
double division:        0.8333333333333333
```

The first line adds tops and bottoms: $\frac{1+1}{2+3} = \frac{2}{5}$. It looks tidy but is simply false — a half plus a third is more than a half, and $\frac{2}{5}$ is less. The second line is the famous *integer division* trap of Chapter 3: in Java, `1 / 2` between two `int` values throws away the remainder and gives `0`, and so does `1 / 3`, so the "sum" is `0`. The third line casts to `double` first and gets a number that is close to the truth, 0.8333 …, but it is not what was asked: the problem wants two whole numbers, and the infinite decimal $0.8\overline{3}$ cannot even be stored exactly. **Listing 4.5** makes the same mistakes in Python, where `//` is integer (floor) division and `/` always produces a floating-point number.

Listing 4.5 `wrong_fraction_sum.py`: the same three mistakes in Python.

```
"""Three tempting but wrong ways to add 1/2 and 1/3."""
e1, d1, e2, d2 = 1, 2, 1, 3
print(f"tops and bottoms added: {e1 + e2}/{d1 + d2}")
print(f"floor division:         {e1 // d1 + e2 // d2}")
print(f"true division:          {e1 / d1 + e2 / d2}")
```

```
tops and bottoms added: 2/5
floor division:         0
true division:          0.8333333333333333
```

A debugger shows the second mistake at once: step over `e1 / d1` and the watch window shows `0`, not `0.5`.

> **CHECK YOURSELF.** Add two methods `differenceNumerator(e1, d1, e2, d2)` and `differenceDenominator(d1, d2)` for $\frac{e_1}{d_1} - \frac{e_2}{d_2}$. What must `printDifference(1, 3, 3, 4)` print?
>
> *Answer.* By the same common-bottom argument, $\frac{e_1}{d_1} - \frac{e_2}{d_2} = \frac{e_1 d_2 - e_2 d_1}{d_1 d_2}$, so `differenceNumerator` returns `e1 * d2 - e2 * d1` and `differenceDenominator` returns `d1 * d2`. For $\frac{1}{3} - \frac{3}{4}$: $1 \cdot 4 - 3 \cdot 3 = 4 - 9 = -5$ and $3 \cdot 4 = 12$, so the line is `1/3 - 3/4 = -5/12`.

4.4 Problem 4.2 — Analog Clock Angle

4.4.1 The problem

> **PROBLEM 4.2.** The time is 13:03. We represent the time by two variables, `hours = 13;` and `minutes = 3;`. Write a Java class `Q12Time` that computes the angle between the hour hand and the minute hand on a traditional analog clock. Angles are measured counterclockwise from the hour hand to the minute hand. The result is rounded and normalized so that it lies between 0 and 359 inclusive. For instance, the angles at 3:00 and at 9:00 are 90° and 270° (**Figure 4.2**). Hint: one minute corresponds to 6° and one hour to 30°; start at 12 o'clock. The program must work for any input with `hours` from 0 to 23 and `minutes` from 0 to 59, and its comments should explain why it is correct. Test it for 9:00, 3:00, 18:00, 1:00, 2:30 and 4:41, whose results are 270, 90, 180, 30, 255 and 254.

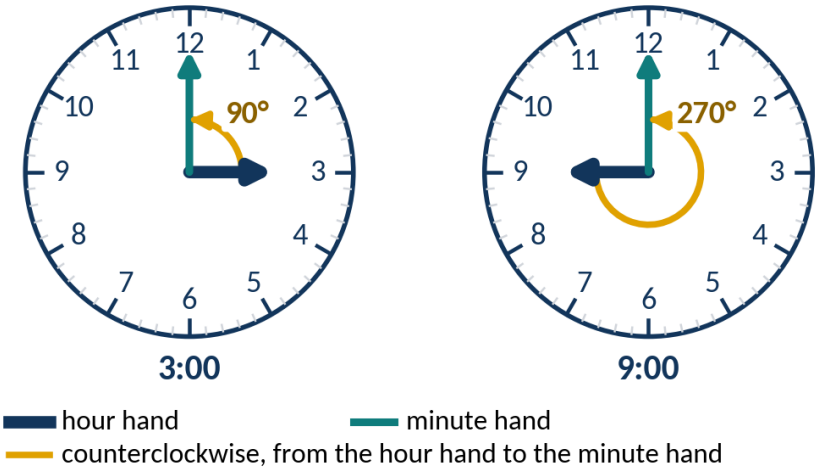


Figure 4.2 The clock at 3:00 (left) and at 9:00 (right); the amber arc runs counterclockwise from the hour hand (short, dark) to the minute hand (long, teal), giving 90° and 270°.

4.4.2 Understanding it by hand

Both hands turn *clockwise*, and we measure where each one points as an angle clockwise from the 12. The minute hand goes all the way round, 360°, in 60 minutes, so it turns $360/60 = 6°$ per minute. The hour hand goes round in 12 hours, $360/12 = 30°$ per hour — but it does not jump from one number to the next on the hour; it creeps forward continuously, so after m minutes it has moved an extra $\frac{m}{60}$ of an hour. With h the hour and m the minute:

$$\theta_{\text{hour}} = 30\left(h \bmod 12 + \frac{m}{60}\right) \tag{4.3}$$

$$\theta_{\text{minute}} = 6m \tag{4.4}$$

Equation (4.3) is the hour hand and Equation (4.4) the minute hand. Here $h \bmod 12$ is the remainder of h after division by 12, which turns 18 o'clock into 6 o'clock. Now the direction. Going *counterclockwise* from the hour hand means going *backwards* on the dial, so the angle from the hour hand to the minute hand is the hour hand's position *minus* the minute hand's position. That difference may be negative, and may be up to 360° away from the range we want, so we bring it into [0,360) by taking it modulo 360, and only then round:

$$\alpha = \text{round}((\theta_{\text{hour}} - \theta_{\text{minute}}) \bmod 360) \bmod 360 \tag{4.5}$$

The last $\bmod 360$ in Equation (4.5) catches one rare case, which we meet below. **Table 4.2** applies Equation (4.3) to Equation (4.5) to the six test times, one column per step.

Table 4.2 The six test times worked by hand: hour hand, minute hand, their difference, the difference brought into [0,360), and the rounded result.

Time	θ_{hour}	θ_{minute}	Difference	mod 360	Result
9:00	$30 \cdot 9 = 270$	0	270	270	270
3:00	$30 \cdot 3 = 90$	0	90	90	90
18:00	$30 \cdot 6 = 180$	0	180	180	180
1:00	$30 \cdot 1 = 30$	0	30	30	30
2:30	$30 \cdot 2.5 = 75$	180	−105	255	255
4:41	$120 + 20.5 = 140.5$	246	−105.5	254.5	254

The last row is the interesting one (**Figure 4.3**). At 4:41 the exact angle is 254.5°, exactly halfway between 254 and 255, and the expected answer is 254. So "rounded" here cannot mean the school rule "a half always rounds up", which would give 255. It means *round half to even*: a value exactly halfway between two whole numbers goes to the *even* one. This is the default rounding of the IEEE 754 floating-point standard [4], it is what Python's built-in `round` does [5], and in Java it is `Math.rint` [6]. Is a half likely? Yes: the difference is $30h - 5.5m$, and whenever m is odd that ends in .5. So half the minutes of every hour are a tie, and the rounding rule matters constantly.

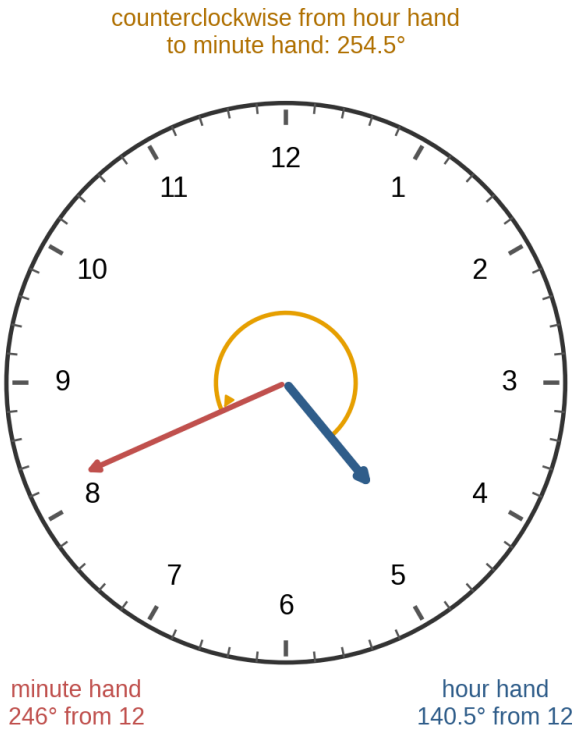


Figure 4.3 The clock at 4:41: the hour hand has crept to 140.5°, the minute hand points at 246°, and the counterclockwise angle between them is 254.5°, which rounds half to even to 254.

Two edge examples of our own. At 2:11, $\theta_{\text{hour}} = 60 + 5.5 = 65.5$ and $\theta_{\text{minute}} = 66$; the difference is -0.5, modulo 360 it is 359.5, and 359.5 rounds (half to even) to 360 — which is outside $0..359$. The final mod 360 turns it into 0, the right answer, since the hands almost coincide. At 23:59, $\theta_{\text{hour}} = 30 \cdot \left(11 + \frac{59}{60}\right) = 359.5$ and $\theta_{\text{minute}} = 354$, the difference is 5.5, and 5.5 rounds half to even to 6.

4.4.3 Classes, fields and methods

Again there is no object with a memory, only a computation, split into the three quantities of Equation (4.3) to Equation (4.5). Two named constants replace the "magic numbers" 30 and 6.

```
+-------------------------------------------------------------+
| Q12Time                                                     |
+-------------------------------------------------------------+
| + DEGREES_PER_HOUR : double = 30.0      (constant)          |
| + DEGREES_PER_MINUTE : double = 6.0     (constant)          |
+-------------------------------------------------------------+
| + hourHandAngle(hours, minutes) : double   (static)         |
| + minuteHandAngle(minutes) : double        (static)         |
| + angle(hours, minutes) : int              (static)         |
| + printAngle(hours, minutes), main(args)                    |
+-------------------------------------------------------------+
```

4.4.4 The Java solution

Listing 4.6 is the class, with the reasoning above written into its comments, as the problem asks.

Listing 4.6 `Q12Time.java`: the counterclockwise angle from the hour hand to the minute hand.

```
/**
 * Problem 4.2: the angle, measured counterclockwise from the hour hand
 * to the minute hand, on an analog clock.
 * The hour hand turns 360/12 = 30 degrees per hour, and also moves on
 * as the minutes pass; the minute hand turns 360/60 = 6 degrees per minute.
 * Both angles are measured clockwise from 12 o'clock, so the counterclockwise
 * angle from the hour hand to the minute hand is (hour angle - minute angle).
 */
public class Q12Time {
    public static final double DEGREES_PER_HOUR = 30.0;
    public static final double DEGREES_PER_MINUTE = 6.0;

    public static double hourHandAngle(int hours, int minutes) {
        return DEGREES_PER_HOUR * (hours % 12 + minutes / 60.0);
    }

    public static double minuteHandAngle(int minutes) {
        return DEGREES_PER_MINUTE * minutes;
    }

    public static int angle(int hours, int minutes) {
        double difference = hourHandAngle(hours, minutes) - minuteHandAngle(minutes);
        double normalized = (difference % 360 + 360) % 360; // 0 <= normalized < 360
        int rounded = (int) Math.rint(normalized);          // halves go to the even
neighbor
        return rounded % 360;                               // 359.5 rounds to 360, which
is 0
    }

    public static void printAngle(int hours, int minutes) {
        System.out.println("At " + hours + ":" + (minutes < 10 ? "0" : "") + minutes
                + " the hour hand is at " + hourHandAngle(hours, minutes)
                + ", the minute hand at " + minuteHandAngle(minutes)
                + "; the angle is " + angle(hours, minutes) + " degrees.");
    }

    public static void main(String[] args) {
        printAngle(9, 0);
        printAngle(3, 0);
        printAngle(18, 0);
        printAngle(1, 0);
        printAngle(2, 30);
        printAngle(4, 41);
    }
}
```

```
At 9:00 the hour hand is at 270.0, the minute hand at 0.0; the angle is 270 degrees.
At 3:00 the hour hand is at 90.0, the minute hand at 0.0; the angle is 90 degrees.
At 18:00 the hour hand is at 180.0, the minute hand at 0.0; the angle is 180 degrees.
At 1:00 the hour hand is at 30.0, the minute hand at 0.0; the angle is 30 degrees.
At 2:30 the hour hand is at 75.0, the minute hand at 180.0; the angle is 255 degrees.
At 4:41 the hour hand is at 140.5, the minute hand at 246.0; the angle is 254 degrees.
```

The printed hand positions are the second and third columns of **Table 4.2**, and the angles are the last column. **Listing 4.7** checks all six given times and the three edge times.

Listing 4.7 `Q12TimeTest.java`: the six given times and three edge cases.

```
/** Plain test driver for Q12Time: run it with "java Q12TimeTest". */
public class Q12TimeTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        check("9:00", 270, Q12Time.angle(9, 0));
        check("3:00", 90, Q12Time.angle(3, 0));
        check("18:00", 180, Q12Time.angle(18, 0));
        check("1:00", 30, Q12Time.angle(1, 0));
        check("2:30", 255, Q12Time.angle(2, 30));
        check("4:41", 254, Q12Time.angle(4, 41));
        check("0:00", 0, Q12Time.angle(0, 0));
        check("2:11", 0, Q12Time.angle(2, 11));
        check("23:59", 6, Q12Time.angle(23, 59));
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

```
PASS 9:00
PASS 3:00
PASS 18:00
PASS 1:00
PASS 2:30
PASS 4:41
PASS 0:00
PASS 2:11
PASS 23:59
9 passed, 0 failed
```

4.4.5 The Python solution

Listing 4.8 `q12_time.py`: Problem 4.2 in Python, where `%` and `round` already behave as Equation (4.5) requires.

```
"""Problem 4.2: the angle, measured counterclockwise from the hour hand
to the minute hand, on an analog clock (see Q12Time.java for the reasoning).
"""
import unittest

DEGREES_PER_HOUR = 30.0
DEGREES_PER_MINUTE = 6.0


def hour_hand_angle(hours, minutes):
    return DEGREES_PER_HOUR * (hours % 12 + minutes / 60)


def minute_hand_angle(minutes):
    return DEGREES_PER_MINUTE * minutes


def angle(hours, minutes):
    difference = hour_hand_angle(hours, minutes) - minute_hand_angle(minutes)
    normalized = difference % 360   # Python's % already gives 0 <= normalized < 360
    return round(normalized) % 360  # round() sends halves to the even neighbor


def print_angle(hours, minutes):
    print(f"At {hours}:{minutes:02d} the hour hand is at {hour_hand_angle(hours, minutes)},
"
          f"the minute hand at {minute_hand_angle(minutes)}; "
          f"the angle is {angle(hours, minutes)} degrees.")


class TestQ12Time(unittest.TestCase):
    def test_given_times(self):
        self.assertEqual(angle(9, 0), 270)
        self.assertEqual(angle(3, 0), 90)
        self.assertEqual(angle(18, 0), 180)
        self.assertEqual(angle(1, 0), 30)
        self.assertEqual(angle(2, 30), 255)
        self.assertEqual(angle(4, 41), 254)

    def test_edges(self):
        self.assertEqual(angle(0, 0), 0)
        self.assertEqual(angle(2, 11), 0)
        self.assertEqual(angle(23, 59), 6)
```

Listing 4.8 (Continued) *q12_time.py: Problem 4.2 in Python, where % and round already behave as Equation (4.5) requires.*

```
if __name__ == "__main__":
    for h, m in [(9, 0), (3, 0), (18, 0), (1, 0), (2, 30), (4, 41)]:
        print_angle(h, m)
    unittest.main(verbosity=2)
```

```
At 9:00 the hour hand is at 270.0, the minute hand at 0.0; the angle is 270 degrees.
At 3:00 the hour hand is at 90.0, the minute hand at 0.0; the angle is 90 degrees.
At 18:00 the hour hand is at 180.0, the minute hand at 0.0; the angle is 180 degrees.
At 1:00 the hour hand is at 30.0, the minute hand at 0.0; the angle is 30 degrees.
At 2:30 the hour hand is at 75.0, the minute hand at 180.0; the angle is 255 degrees.
At 4:41 the hour hand is at 140.5, the minute hand at 246.0; the angle is 254 degrees.
test_edges (__main__.TestQ12Time.test_edges) ... ok
test_given_times (__main__.TestQ12Time.test_given_times) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Listing 4.8 is shorter than **Listing 4.6** in one place, and the reason is worth knowing.

4.4.6 Line by line

- `minutes / 60.0` in Java — the `.0` makes the division a `double` division. Writing `minutes / 60` would divide two `int` values and give `0` for every minute below 60, freezing the hour hand on the hour. In Python `/` always divides exactly, so `minutes / 60` is correct there.
- `(difference % 360 + 360) % 360` — in Java the remainder operator `%` keeps the sign of the *left* operand: `-105.0 % 360` is `-105.0` [7]. Adding 360 makes it positive, and the second `% 360` removes the extra 360 again when the value was already positive. In Python, `%` with a positive right operand always returns a value between 0 and the divisor [5], so `difference % 360` alone is `255.0`. This one difference between the two languages causes more wrong answers than any other in this chapter.
- `(int) Math.rint(normalized)` — `Math.rint` returns the whole number nearest to its argument, choosing the even one on a tie, as a `double` such as `254.0`; the *cast* `(int)` turns it into the `int` 254. Python's `round(normalized)` does both at once and returns an `int`.
- `return rounded % 360;` — the 2:11 case: `359.5` rounds to `360`, and `360 % 360` is `0`.
- `(minutes < 10 ? "0" : "") + minutes` — the *conditional operator* `condition ? a : b` from Chapter 3 chooses `"0"` or nothing, so that 9 o'clock prints as `9:00` and not `9:0`. Python writes the same thing as the format `{minutes:02d}`: at least two digits, padded with zeros.
- `hours % 12` is, strictly, not needed, because the last modulo 360 would absorb the extra $12 \cdot 30 = 360$ degrees. We keep it because it states the physical truth — a clock face has 12 hours — and makes the printed hand position `180.0` rather than `540.0` at 18:00.

4.4.7 A tempting wrong solution

Listing 4.9 shows the three most tempting slips at 4:41, whose correct answer is 254.

Listing 4.9 `WrongClockAngle.java`: three tempting slips in the clock angle at 4:41.

```
/** Three tempting but wrong versions of the clock angle at 4:41 (correct: 254). */
public class WrongClockAngle {
    public static void main(String[] args) {
        int hours = 4, minutes = 41;
        double difference = 30 * (hours % 12 + minutes / 60.0) - 6 * minutes;
        System.out.println("difference:        " + difference);
        System.out.println("no +360:           " + Math.round(difference) % 360);
        System.out.println("Math.round:        " + (Math.round(difference) + 360) % 360);
        int frozen = 30 * (hours % 12 + minutes / 60) - 6 * minutes;
        System.out.println("minutes / 60 int:  " + (frozen % 360 + 360) % 360);
    }
}
```

```
difference:        -105.5
no +360:           -105
Math.round:        255
minutes / 60 int:  234
```

Forgetting the `+ 360` leaves a negative angle, −105. Using `Math.round`, which rounds halves *up* (towards positive infinity), turns −105.5 into −105 and finally gives 255, one degree off. And writing `minutes / 60` with two `int` values freezes the hour hand at 120°, giving 234. **Listing 4.10** is the Python version, and it contains a warning of a different kind.

Listing 4.10 `wrong_clock_angle.py`: in Python, rounding before normalizing happens to give the right answer.

```
"""Three versions of the clock angle at 4:41 (correct: 254); one is right only by luck."""
import math

hours, minutes = 4, 41
difference = 30 * (hours % 12 + minutes / 60) - 6 * minutes
print(f"difference:        {difference}")
print(f"round, then %:     {round(difference) % 360}")
print(f"round half up:     {math.floor(difference + 0.5) % 360}")
frozen = 30 * (hours % 12 + minutes // 60) - 6 * minutes
print(f"minutes // 60:     {frozen % 360}")
```

```
difference:        -105.5
round, then %:     254
round half up:     255
minutes // 60:     234
```

The second line prints 254, the right answer — but by luck: `round(-105.5)` is −106 because −106 is even, and −106 mod 360 = 254. The code rounds *before* normalizing, which is the wrong order; it only agrees with Equation (4.5) because shifting by 360 does not change which neighbor is even. A program that is right by accident is a program that will be wrong after the next small change, which is why every line of a solution should have a reason you can say aloud.

> **CHECK YOURSELF.** Write a method `clockwiseAngle(hours, minutes)` that measures the angle *clockwise* from the hour hand to the minute hand. What does it return at 4:41 and at 3:00?
>
> *Answer.* Clockwise is the other way round the dial, so the angle is $\theta_{\text{minute}} - \theta_{\text{hour}}$, normalized and rounded the same way; equivalently it is $(360 - \alpha) \bmod 360$ before rounding. At 4:41: $246 - 140.5 = 105.5$, which rounds half to even to 106. At 3:00: $0 - 90 = -90$, normalized to 270. (Check: $254.5 + 105.5 = 360$ and $90 + 270 = 360$.)

4.5 Problem 4.3 — Computer Account

4.5.1 The problem

> **PROBLEM 4.3.** Define a class `Q13ComputerAccount` and a constructor to create it. A computer account is built from three strings, `realName`, `userName` and `password`. It implements the methods `printRealName`, `printUserName`, `printPassword` and `changePassword`. The first three take no arguments. The last takes one argument of type `String`, the new password, and sets the password to it; its return type is `void`. Comment and test the program appropriately, in a separate class `Q13ComputerAccountTest`.

4.5.2 Understanding it by hand

This is the first problem with a real object: something that *remembers*. An account for John Smith, user name `JXS`, password `sesame`, keeps those three strings for as long as the object lives. If we then call `changePassword("senoJ yraM")`, a later `printPassword()` must print the new password, not the old one. And a second account, for Mary Jones, has its *own* three strings: changing John's password must not touch Mary's. On paper, the story of the test is a small table of states:

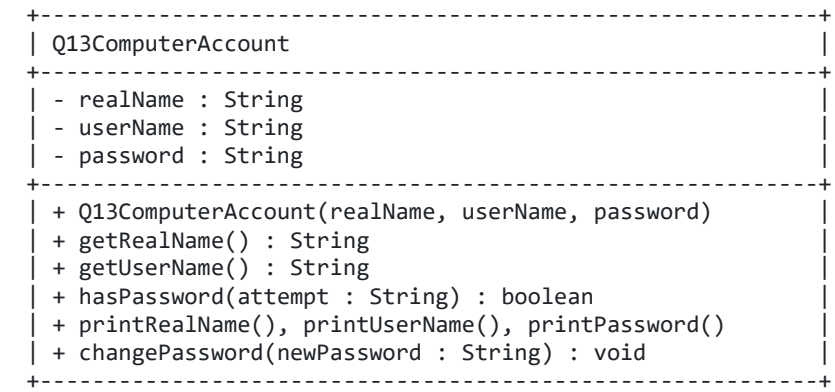
Table 4.3 The passwords of two accounts after each step of the test; the last row shows that each object has its own fields.

Step	John's password	Mary's password
create John with `sesame`	`sesame`	(no Mary yet)
`john.changePassword("senoJ yraM")`	`senoJ yraM`	(no Mary yet)
create Mary with `senoJ yraM`	`senoJ yraM`	`senoJ yraM`
`john.changePassword("sesame")`	`sesame`	`senoJ yraM`

The last row of **Table 4.3** is the whole point of objects: one class, many independent objects.

4.5.3 Classes, fields and methods

Nouns: *account* (the class), *real name*, *user name*, *password* (three `String` fields). Verbs: *print* each field, *change* the password. We add three small methods the problem does not ask for but the tests need: `getRealName` and `getUserName` return a field's value so a test can compare it, and `hasPassword(attempt)` answers "is this the password?" with `true` or `false`, so that we can test `changePassword` without *reading* the password.



The fields are private (the minus signs): the only way in is through the methods. That is *encapsulation*, and it is why `hasPassword` exists — there is deliberately no `getPassword`.

4.5.4 The Java solution

Listing 4.11 is the class and **Listing 4.12** its test driver.

Listing 4.11 `Q13ComputerAccount.java`: an account with private fields and a controlled way to check the password.

```java
/**
 * Problem 4.3: a computer account built from a real name, a user name
 * and a password. The fields are private, so the only way to read or
 * change them is through the methods below.
 */
public class Q13ComputerAccount {
    private String realName;
    private String userName;
    private String password;

    public Q13ComputerAccount(String realName, String userName, String password) {
        this.realName = realName;
        this.userName = userName;
        this.password = password;
    }

    public String getRealName() {
        return realName;
    }

    public String getUserName() {
        return userName;
    }

    // Answers "is this the password?" without ever handing the password out.
    public boolean hasPassword(String attempt) {
        return password.equals(attempt);
    }

    public void printRealName() {
        System.out.println(realName);
    }

    public void printUserName() {
        System.out.println(userName);
    }

    // Required by the problem; a real system would never print a password.
    public void printPassword() {
        System.out.println(password);
    }

    public void changePassword(String newPassword) {
        password = newPassword;
    }
}
```

Listing 4.12 `Q13ComputerAccountTest.java`: the story of **Table 4.3** as automatic checks.

```
/** Plain test driver for Q13ComputerAccount: run it with "java Q13ComputerAccountTest". */
public class Q13ComputerAccountTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        Q13ComputerAccount john = new Q13ComputerAccount("John Smith", "JXS", "sesame");
        john.printRealName();
        john.printUserName();
        john.printPassword();
        john.changePassword("senoJ yraM");
        john.printPassword();

        check("real name", "John Smith", john.getRealName());
        check("user name", "JXS", john.getUserName());
        check("new password accepted", true, john.hasPassword("senoJ yraM"));
        check("old password rejected", false, john.hasPassword("sesame"));

        Q13ComputerAccount mary = new Q13ComputerAccount("Mary Jones", "MYJ", "senoJ
yraM");
        check("second account is separate", "MYJ", mary.getUserName());
        john.changePassword("sesame");
        check("changing John leaves Mary alone", true, mary.hasPassword("senoJ yraM"));
        String typed = new String("sesame");
        check("equals compares characters", true, john.hasPassword(typed));
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

Compile both with `javac Q13ComputerAccount.java Q13ComputerAccountTest.java` and run the driver with `java Q13ComputerAccountTest`:

```
John Smith
JXS
sesame
senoJ yraM
PASS real name
PASS user name
PASS new password accepted
PASS old password rejected
PASS second account is separate
PASS changing John leaves Mary alone
PASS equals compares characters
7 passed, 0 failed
```

The four printed lines come from the four `print…` calls; the checks follow **Table 4.3** row by row.

4.5.5 The Python solution

Listing 4.13 follows the same class box; in Python the getters and the test class live in one file.

Listing 4.13 `q13_computer_account.py`: Problem 4.3 in Python.

```
"""Problem 4.3: a computer account built from a real name, a user name
and a password. The leading underscore marks each attribute as private
by convention: code outside the class should use the methods instead.
"""
import unittest


class Q13ComputerAccount:
    def __init__(self, real_name, user_name, password):
        self._real_name = real_name
        self._user_name = user_name
        self._password = password

    def get_real_name(self):
        return self._real_name

    def get_user_name(self):
        return self._user_name

    def has_password(self, attempt):
        # Answers "is this the password?" without ever handing the password out.
        return self._password == attempt

    def print_real_name(self):
        print(self._real_name)

    def print_user_name(self):
        print(self._user_name)

    def print_password(self):
        # Required by the problem; a real system would never print a password.
        print(self._password)

    def change_password(self, new_password):
        self._password = new_password


class TestQ13ComputerAccount(unittest.TestCase):
    def setUp(self):
        self.john = Q13ComputerAccount("John Smith", "JXS", "sesame")

    def test_names(self):
        self.assertEqual(self.john.get_real_name(), "John Smith")
        self.assertEqual(self.john.get_user_name(), "JXS")
```

Listing 4.13 (Continued) *q13_computer_account.py: Problem 4.3 in Python.*

```
    def test_change_password(self):
        self.john.change_password("senoJ yraM")
        self.assertTrue(self.john.has_password("senoJ yraM"))
        self.assertFalse(self.john.has_password("sesame"))

    def test_accounts_are_separate(self):
        mary = Q13ComputerAccount("Mary Jones", "MYJ", "senoJ yraM")
        self.john.change_password("open")
        self.assertTrue(mary.has_password("senoJ yraM"))

    def test_typed_text_compares_by_characters(self):
        typed = "".join(["ses", "ame"])
        self.assertTrue(self.john.has_password(typed))


if __name__ == "__main__":
    john = Q13ComputerAccount("John Smith", "JXS", "sesame")
    john.print_real_name()
    john.print_user_name()
    john.print_password()
    john.change_password("senoJ yraM")
    john.print_password()
    unittest.main(verbosity=2)
```

```
John Smith
JXS
sesame
senoJ yraM
test_accounts_are_separate (__main__.TestQ13ComputerAccount.test_accounts_are_separate) ...
ok
test_change_password (__main__.TestQ13ComputerAccount.test_change_password) ... ok
test_names (__main__.TestQ13ComputerAccount.test_names) ... ok
test_typed_text_compares_by_characters
(__main__.TestQ13ComputerAccount.test_typed_text_compares_by_characters) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

4.5.6 Line by line

- `private String realName;` — each object gets its own copy of the three fields; `private` means that only code inside `Q13ComputerAccount` may name them.
- `this.realName = realName;` — inside the constructor, the parameter `realName` hides the field of the same name. `this.realName` means "the field of the object being built", so the line copies the parameter into the field. Forgetting `this.` is a classic silent bug: `realName = realName;` assigns the parameter to itself, and the field stays `null`.
- `return password.equals(attempt);` — `equals` compares two strings character by character. This is the only correct way to compare string *contents* in Java; see the tempting wrong solution below.
- `public void changePassword(String newPassword) { password = newPassword; }` — no `this.` is needed here, because no parameter is called `password`.
- `String typed = new String("sesame");` in the test deliberately creates a *second* string object with the same characters, as happens when text is typed at a keyboard or read from a file. The check proves that `hasPassword` compares characters, not objects.

- In Python the fields are created by assignment in `__init__`, the method Python calls when an object is created. The leading underscore of `self._password` says "private, please" — a *convention* that Python does not enforce (Problem 4.7 shows what that means). The `setUp` method of the test class runs before *every* test, so each test starts with a fresh John.

Two design remarks. First, the problem asks for `printPassword`, so we provide it; a real system would never print a password, and one could reasonably make that method private. Second, `changePassword` here does not ask for the old password, because the problem does not require it; the *check yourself* adds that safety.

4.5.7 A tempting wrong solution

The tempting shortcut is to compare strings with `==`, as one compares numbers. **Listing 4.14** shows why that fails.

Listing 4.14 `WrongPasswordCheck.java`: `==` compares object identity, `equals` compares characters.

```
/** Comparing strings with == asks "same object?", not "same characters?". */
public class WrongPasswordCheck {
    public static void main(String[] args) {
        String stored = "sesame";
        String typed = new String("sesame"); // as if it had been typed at the keyboard
        System.out.println("typed == stored:      " + (typed == stored));
        System.out.println("typed.equals(stored): " + typed.equals(stored));
    }
}
```

```
typed == stored:      false
typed.equals(stored): true
```

For objects, `==` asks "are these two *the same object*?" [7]. `stored` and `typed` are two different objects that happen to hold the same six characters, so `==` answers `false`, and a user who types the right password is refused. Worse, `==` sometimes *seems* to work — when both strings are literals written in the program, Java shares one object between them — so the bug hides during quick tests and appears only with real input. Python's `==` compares contents, and its `is` is the identity test, as **Listing 4.15** shows.

Listing 4.15 `wrong_password_check.py`: in Python, `is` is the identity test and `==` compares contents.

```
"""Comparing strings with 'is' asks "same object?", not "same characters?"."""
stored = "sesame"
typed = "".join(["ses", "ame"])  # as if it had been typed at the keyboard
print(f"typed is stored: {typed is stored}")
print(f"typed == stored: {typed == stored}")
```

```
typed is stored: False
typed == stored: True
```

> **CHECK YOURSELF.** Change `changePassword` so that it takes two arguments, the old and the new password, changes the password only if the old one is right, and returns `true` or `false` to say whether it did. What do `changePassword("wrong", "x")` and then `changePassword("sesame", "x")` return for a new account with password `sesame`?
>
> *Answer.* In Java: `public boolean changePassword(String oldPassword, String newPassword) { if (!password.equals(oldPassword)) { return false; } password = newPassword; return true; }`. The first call returns `false` and leaves the password `sesame`;

the second returns `true` and the password becomes `x`. In Python the body is `if self._password != old_password: return False`, then the assignment and `return True`.

4.6 Problem 4.4 — Complex Numbers

4.6.1 The problem

PROBLEM 4.4. Define a class `Q14ComplexNumber` with a constructor and getter methods. A `Q14ComplexNumber` represents a complex number with a real and an imaginary part. Implement methods to (1) add two complex numbers, (2) subtract two complex numbers, (3) multiply two complex numbers, and (4) convert the complex number to a string `"a + bi"`, where a is the real and b the imaginary part. The operations are $(a + bi) + (c + di) = (a + c) + (b + d)i$, $(a + bi) - (c + di) = (a - c) + (b - d)i$ and Equation (4.6). Test the class in `Q14ComplexNumberTest`.

$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i \tag{4.6}$$

4.6.2 Understanding it by hand

A complex number is a pair of ordinary numbers, a *real part* a and an *imaginary part* b, written $a + bi$, where i is a number whose square is -1. You do not need to know more mathematics than that: the three formulas tell us exactly what to compute. Equation (4.6) comes from multiplying out the brackets as usual — $ac + adi + bci + bd\,i^2$ — and replacing i^2 by -1, which moves bd into the real part with a minus sign.

Take $c_1 = 1 + 2i$ and $c_2 = 3 + 4i$:

- sum: $(1 + 3) + (2 + 4)i = 4 + 6i$;
- difference: $(1 - 3) + (2 - 4)i = -2 - 2i$;
- product: real part $1 \cdot 3 - 2 \cdot 4 = 3 - 8 = -5$, imaginary part $1 \cdot 4 + 2 \cdot 3 = 4 + 6 = 10$, so $-5 + 10i$;
- edge example: $i \cdot i$, that is $(0 + 1i)(0 + 1i)$: real part $0 \cdot 0 - 1 \cdot 1 = -1$, imaginary part $0 \cdot 1 + 1 \cdot 0 = 0$, so $-1 + 0i$, as the definition of i demands.

The difference raises a question the statement does not answer: should $-2 - 2i$ be printed as `-2.0 + -2.0i`, which follows the pattern `"a + bi"` literally, or as `-2.0 - 2.0i`, which is how any person writes it? We choose the second, using an `if`, and our tests say so.

4.6.3 Classes, fields and methods

```
+-----------------------------------------------------------+
| Q14ComplexNumber                                          |
+-----------------------------------------------------------+
| - real : double                                           |
| - imaginary : double                                      |
+-----------------------------------------------------------+
| + Q14ComplexNumber(real, imaginary)                       |
| + getReal() : double                                      |
| + getImaginary() : double                                 |
| + add(other : Q14ComplexNumber) : Q14ComplexNumber        |
| + subtract(other : Q14ComplexNumber) : Q14ComplexNumber   |
| + multiply(other : Q14ComplexNumber) : Q14ComplexNumber   |
| + toString() : String                                     |
+-----------------------------------------------------------+
```

The key design decision is in the return types: `add` takes *another* complex number and returns a *new* one. It does not change `this`, and it does not change `other`. Then `c1.add(c2)` behaves like $c_1 + c_2$ in mathematics, where adding does not change the numbers you add.

4.6.4 The Java solution

Listing 4.16 is the class; **Listing 4.17** checks the four hand examples and that `c1` is unchanged after being used.

Listing 4.16 `Q14ComplexNumber.java`: complex numbers whose operations return new objects.

```
/**
 * Problem 4.4: a complex number a + bi with real part a and imaginary part b.
 *   (a + bi) + (c + di) = (a + c) + (b + d)i
 *   (a + bi) - (c + di) = (a - c) + (b - d)i
 *   (a + bi) * (c + di) = (ac - bd) + (ad + bc)i
 * Every operation returns a new object and leaves both operands unchanged.
 */
public class Q14ComplexNumber {
    private double real;
    private double imaginary;

    public Q14ComplexNumber(double real, double imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    public double getReal() {
        return real;
    }

    public double getImaginary() {
        return imaginary;
    }

    public Q14ComplexNumber add(Q14ComplexNumber other) {
        return new Q14ComplexNumber(real + other.real, imaginary + other.imaginary);
    }

    public Q14ComplexNumber subtract(Q14ComplexNumber other) {
        return new Q14ComplexNumber(real - other.real, imaginary - other.imaginary);
    }

    public Q14ComplexNumber multiply(Q14ComplexNumber other) {
        double newReal = real * other.real - imaginary * other.imaginary;
        double newImaginary = real * other.imaginary + imaginary * other.real;
        return new Q14ComplexNumber(newReal, newImaginary);
    }

    @Override
    public String toString() {
        if (imaginary < 0) {
            return real + " - " + (-imaginary) + "i";
        }
        return real + " + " + imaginary + "i";
    }
}
```

Listing 4.17 `Q14ComplexNumberTest.java`: the four hand examples and a check that the operands are unchanged.

```java
/** Plain test driver for Q14ComplexNumber: run it with "java Q14ComplexNumberTest". */
public class Q14ComplexNumberTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        Q14ComplexNumber c1 = new Q14ComplexNumber(1, 2);
        Q14ComplexNumber c2 = new Q14ComplexNumber(3, 4);
        System.out.println("Sum: (" + c1 + ") + (" + c2 + ") = " + c1.add(c2));
        System.out.println("Difference: (" + c1 + ") - (" + c2 + ") = " + c1.subtract(c2));
        System.out.println("Product: (" + c1 + ") * (" + c2 + ") = " + c1.multiply(c2));

        check("sum", "4.0 + 6.0i", c1.add(c2).toString());
        check("difference", "-2.0 - 2.0i", c1.subtract(c2).toString());
        check("product", "-5.0 + 10.0i", c1.multiply(c2).toString());
        check("i * i = -1", "-1.0 + 0.0i",
                new Q14ComplexNumber(0, 1).multiply(new Q14ComplexNumber(0,
1)).toString());
        check("operand unchanged", "1.0 + 2.0i", c1.toString());
        check("getReal", 3.0, c2.getReal());
        check("getImaginary", 4.0, c2.getImaginary());
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

```
Sum: (1.0 + 2.0i) + (3.0 + 4.0i) = 4.0 + 6.0i
Difference: (1.0 + 2.0i) - (3.0 + 4.0i) = -2.0 - 2.0i
Product: (1.0 + 2.0i) * (3.0 + 4.0i) = -5.0 + 10.0i
PASS sum
PASS difference
PASS product
PASS i * i = -1
PASS operand unchanged
PASS getReal
PASS getImaginary
7 passed, 0 failed
```

4.6.5 The Python solution

In **Listing 4.18**, `__str__` plays the role of `toString`.

Listing 4.18 `q14_complex_number.py`: Problem 4.4 in Python, where `__str__` plays the role of `toString`.

```
"""Problem 4.4: a complex number a + bi (formulas as in Q14ComplexNumber.java).
Every operation returns a new object and leaves both operands unchanged.
"""
import unittest


class Q14ComplexNumber:
    def __init__(self, real, imaginary):
        self._real = float(real)            # float, so 1 prints as 1.0, as in Java
        self._imaginary = float(imaginary)

    def get_real(self):
        return self._real

    def get_imaginary(self):
        return self._imaginary

    def add(self, other):
        return Q14ComplexNumber(self._real + other._real, self._imaginary +
other._imaginary)

    def subtract(self, other):
        return Q14ComplexNumber(self._real - other._real, self._imaginary -
other._imaginary)

    def multiply(self, other):
        new_real = self._real * other._real - self._imaginary * other._imaginary
        new_imaginary = self._real * other._imaginary + self._imaginary * other._real
        return Q14ComplexNumber(new_real, new_imaginary)

    def __str__(self):
        if self._imaginary < 0:
            return f"{self._real} - {-self._imaginary}i"
        return f"{self._real} + {self._imaginary}i"


class TestQ14ComplexNumber(unittest.TestCase):
    def setUp(self):
        self.c1 = Q14ComplexNumber(1, 2)
        self.c2 = Q14ComplexNumber(3, 4)

    def test_add(self):
        self.assertEqual(str(self.c1.add(self.c2)), "4.0 + 6.0i")

    def test_subtract(self):
        self.assertEqual(str(self.c1.subtract(self.c2)), "-2.0 - 2.0i")
```

Listing 4.18 (Continued) *q14_complex_number.py: Problem 4.4 in Python, where __str__ plays the role of toString.*

```
    def test_multiply(self):
        self.assertEqual(str(self.c1.multiply(self.c2)), "-5.0 + 10.0i")

    def test_i_squared(self):
        i = Q14ComplexNumber(0, 1)
        self.assertEqual(str(i.multiply(i)), "-1.0 + 0.0i")

    def test_operands_unchanged(self):
        self.c1.add(self.c2)
        self.assertEqual(str(self.c1), "1.0 + 2.0i")
        self.assertEqual(self.c2.get_real(), 3.0)
        self.assertEqual(self.c2.get_imaginary(), 4.0)


if __name__ == "__main__":
    c1 = Q14ComplexNumber(1, 2)
    c2 = Q14ComplexNumber(3, 4)
    print(f"Sum: ({c1}) + ({c2}) = {c1.add(c2)}")
    print(f"Difference: ({c1}) - ({c2}) = {c1.subtract(c2)}")
    print(f"Product: ({c1}) * ({c2}) = {c1.multiply(c2)}")
    unittest.main(verbosity=2)
```

```
Sum: (1.0 + 2.0i) + (3.0 + 4.0i) = 4.0 + 6.0i
Difference: (1.0 + 2.0i) - (3.0 + 4.0i) = -2.0 - 2.0i
Product: (1.0 + 2.0i) * (3.0 + 4.0i) = -5.0 + 10.0i
test_add (__main__.TestQ14ComplexNumber.test_add) ... ok
test_i_squared (__main__.TestQ14ComplexNumber.test_i_squared) ... ok
test_multiply (__main__.TestQ14ComplexNumber.test_multiply) ... ok
test_operands_unchanged (__main__.TestQ14ComplexNumber.test_operands_unchanged) ... ok
test_subtract (__main__.TestQ14ComplexNumber.test_subtract) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

4.6.6 Line by line

- `return new Q14ComplexNumber(real + other.real, imaginary + other.imaginary);` — `other.real` reads a *private* field of another object. That is allowed, because privacy in Java belongs to the *class*, not to the object: code inside `Q14ComplexNumber` may see the private fields of every `Q14ComplexNumber`.
- In `multiply`, two local variables `newReal` and `newImaginary` hold the two halves of Equation (4.6); each is computed from the *old* values, and only then is the new object built.
- `@Override public String toString()` — every Java object has a `toString` method, which Java calls whenever the object is joined to a string with `+`, as in `"Sum: " + c1`. Writing our own replaces the default, which would print something like `Q14ComplexNumber@1b6d3586`. `@Override` asks the compiler to check that we really are replacing an existing method.
- `if (imaginary < 0) { return real + " - " + (-imaginary) + "i"; }` — for a negative imaginary part, print a minus sign and the *positive* value. The parentheses around `-imaginary` matter: without them, `" - " + -imaginary` still works, but a reader has to think about it.
- In Python, `float(real)` in `__init__` makes `Q14ComplexNumber(1, 2)` store `1.0` and `2.0`, so that Python prints `1.0 + 2.0i` exactly as Java does. The method `__str__` is what `print` and f-strings call, the Python twin of `toString`.

4.6.7 A tempting wrong solution

Listing 4.19 shows the two tempting slips: multiplying part by part, and printing the sign naively.

Listing 4.19 `WrongComplex.java`: multiplying part by part, and printing a negative part naively.

```
/** Multiplying part by part forgets that i * i = -1; naive printing shows "+ -". */
public class WrongComplex {
    public static void main(String[] args) {
        double a = 1, b = 2, c = 3, d = 4;
        System.out.println("part by part: " + (a * c) + " + " + (b * d) + "i");
        System.out.println("correct:      " + (a * c - b * d) + " + " + (a * d + b * c) +
"i");
        double re = a - c, im = b - d;
        System.out.println("naive text:   " + re + " + " + im + "i");
    }
}
```

```
part by part: 3.0 + 8.0i
correct:      -5.0 + 10.0i
naive text:   -2.0 + -2.0i
```

Multiplying real by real and imaginary by imaginary feels natural, because that is how addition works — but it forgets the cross terms ad and bc and the fact that $i \cdot i = -1$, so it gives $3 + 8i$ instead of $-5 + 10i$. The naive string shows `-2.0 + -2.0i`, which is not wrong mathematically but is ugly. **Listing 4.20** repeats both in Python.

Listing 4.20 `wrong_complex.py`: the same two slips in Python.

```
"""Multiplying part by part forgets that i * i = -1; naive printing shows "+ -"."""
a, b, c, d = 1.0, 2.0, 3.0, 4.0
print(f"part by part: {a * c} + {b * d}i")
print(f"correct:      {a * c - b * d} + {a * d + b * c}i")
re, im = a - c, b - d
print(f"naive text:   {re} + {im}i")
```

```
part by part: 3.0 + 8.0i
correct:      -5.0 + 10.0i
naive text:   -2.0 + -2.0i
```

> **CHECK YOURSELF.** Add a method `conjugate()` that returns $a - bi$ for $a + bi$. What does `c1.multiply(c1.conjugate())` print for $c_1 = 1 + 2i$, and why is the imaginary part zero?
>
> *Answer.* `conjugate` returns `new Q14ComplexNumber(real, -imaginary)`. Then $(1 + 2i)(1 - 2i)$ has real part $1 \cdot 1 - 2 \cdot (-2) = 1 + 4 = 5$ and imaginary part $1 \cdot (-2) + 2 \cdot 1 = 0$, so it prints `5.0 + 0.0i`. In general $(a + bi)(a - bi) = a^2 + b^2$, because the cross terms $-abi$ and $+abi$ cancel.

4.7 Problem 4.5 — Weight Conversion

4.7.1 The problem

> **PROBLEM 4.5.** Chapter 3 converted masses given in the imperial system into the metric system. Write a program that deals with weights given in pounds, using Equation (4.7). Define a class `Q15Weight` with a constructor `Q15Weight(double p)` that creates a weight given in pounds, and methods `getPounds` and `getKilograms`, which return the weight in pounds and in kilograms. Test the class in `Q15WeightTest`.

$$1 \text{ pound} = 0.45359237 \text{ kilograms} \tag{4.7}$$

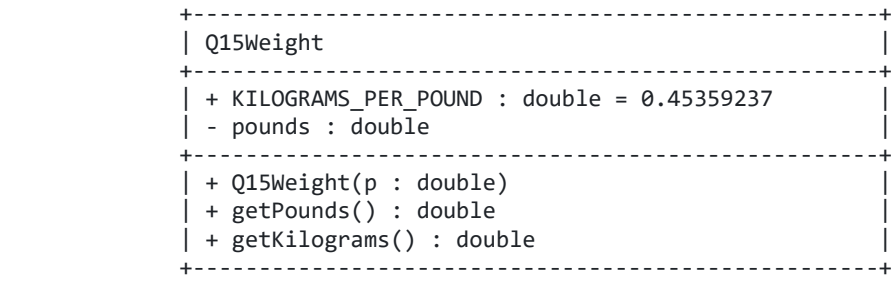
4.7.2 Understanding it by hand

A `Q15Weight` object remembers one weight. Asked for pounds, it returns what it was given; asked for kilograms, it multiplies by 0.45359237. The factor in Equation (4.7) is exact by international agreement, not a measurement, so we write all eight digits. By hand:

- $150 \text{ lb} \times 0.45359237 = 68.0388555 \text{ kg}$;
- $140 \text{ lb} \times 0.45359237 = 63.5029318 \text{ kg}$;
- the person of Chapter 3's converter, 11 stones and 6 pounds, is $11 \cdot 14 + 6 = 160$ pounds, so $160 \times 0.45359237 = 72.5747792$ kg.

The one design question: should the object store both numbers, pounds and kilograms? No. If it stored both, some future method that changes one could forget the other, and the object would contradict itself. It stores the pounds only and *derives* the kilograms whenever they are asked for.

4.7.3 Classes, fields and methods



4.7.4 The Java solution

Listing 4.21 is the whole class: one constant, one field, a constructor and two getters. **Listing 4.22** tests it with the three weights worked by hand.

Listing 4.21 `Q15Weight.java`: a weight stored in pounds, with kilograms computed on demand.

```
/**
 * Problem 4.5: a weight given in pounds. Only the pounds are stored;
 * the kilograms are computed on demand, so the two can never disagree.
 */
public class Q15Weight {
    public static final double KILOGRAMS_PER_POUND = 0.45359237;

    private double pounds;

    public Q15Weight(double p) {
        pounds = p;
    }

    public double getPounds() {
        return pounds;
    }

    public double getKilograms() {
        return pounds * KILOGRAMS_PER_POUND;
    }
}
```

Listing 4.22 `Q15WeightTest.java`: floating-point results are compared with a tolerance.

```java
/** Plain test driver for Q15Weight: run it with "java Q15WeightTest". */
public class Q15WeightTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void checkClose(String label, double expected, double actual) {
        if (Math.abs(expected - actual) < 1e-9) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        Q15Weight john = new Q15Weight(150);
        Q15Weight mary = new Q15Weight(140);
        Q15Weight stones = new Q15Weight(11 * 14 + 6);
        System.out.println("John: " + john.getPounds() + " lb = " + john.getKilograms() + "
kg");
        System.out.println("Mary: " + mary.getPounds() + " lb = " + mary.getKilograms() + "
kg");
        System.out.println("11 st 6 lb: " + stones.getPounds() + " lb = " +
stones.getKilograms() + " kg");

        checkClose("150 lb in pounds", 150.0, john.getPounds());
        checkClose("150 lb in kg", 68.0388555, john.getKilograms());
        checkClose("140 lb in kg", 63.5029318, mary.getKilograms());
        checkClose("1 lb in kg", 0.45359237, new Q15Weight(1).getKilograms());
        checkClose("0 lb in kg", 0.0, new Q15Weight(0).getKilograms());
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

```
John: 150.0 lb = 68.0388555 kg
Mary: 140.0 lb = 63.502931800000006 kg
11 st 6 lb: 160.0 lb = 72.57477920000001 kg
PASS 150 lb in pounds
PASS 150 lb in kg
PASS 140 lb in kg
PASS 1 lb in kg
PASS 0 lb in kg
5 passed, 0 failed
```

Look at Mary's line: `63.502931800000006`, not `63.5029318`. Nothing is wrong. A `double` stores numbers in binary, and 0.45359237, like 0.1, has no exact binary form, so the product carries a tiny error in its sixteenth significant digit [4]. This is why the test driver uses `checkClose`, which accepts a difference smaller than 10^{-9}, instead of `equals`.

4.7.5 The Python solution

Listing 4.23 uses `assertAlmostEqual`, the `unittest` counterpart of `checkClose`.

Listing 4.23 `q15_weight.py`: Problem 4.5 in Python, with `assertAlmostEqual` for the floating-point checks.

```
"""Problem 4.5: a weight given in pounds. Only the pounds are stored;
the kilograms are computed on demand, so the two can never disagree.
"""
import unittest


class Q15Weight:
    KILOGRAMS_PER_POUND = 0.45359237

    def __init__(self, p):
        self._pounds = float(p)

    def get_pounds(self):
        return self._pounds

    def get_kilograms(self):
        return self._pounds * Q15Weight.KILOGRAMS_PER_POUND


class TestQ15Weight(unittest.TestCase):
    def test_pounds_kept(self):
        self.assertEqual(Q15Weight(150).get_pounds(), 150.0)

    def test_kilograms(self):
        self.assertAlmostEqual(Q15Weight(150).get_kilograms(), 68.0388555, places=9)
        self.assertAlmostEqual(Q15Weight(140).get_kilograms(), 63.5029318, places=9)
        self.assertAlmostEqual(Q15Weight(1).get_kilograms(), 0.45359237, places=9)
        self.assertEqual(Q15Weight(0).get_kilograms(), 0.0)


if __name__ == "__main__":
    john = Q15Weight(150)
    mary = Q15Weight(140)
    stones = Q15Weight(11 * 14 + 6)
    print(f"John: {john.get_pounds()} lb = {john.get_kilograms()} kg")
    print(f"Mary: {mary.get_pounds()} lb = {mary.get_kilograms()} kg")
    print(f"11 st 6 lb: {stones.get_pounds()} lb = {stones.get_kilograms()} kg")
    unittest.main(verbosity=2)
```

```
John: 150.0 lb = 68.0388555 kg
Mary: 140.0 lb = 63.502931800000006 kg
11 st 6 lb: 160.0 lb = 72.57477920000001 kg
test_kilograms (__main__.TestQ15Weight.test_kilograms) ... ok
test_pounds_kept (__main__.TestQ15Weight.test_pounds_kept) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Python prints exactly the same digits as Java, because both use the same IEEE 754 `double` [4] and both print the shortest decimal that reads back as the same number.

4.7.6 Line by line

- `public static final double KILOGRAMS_PER_POUND = 0.45359237;` — `static` means one copy for the whole class, `final` that it can never change. Constants are named in capitals with underscores, so a reader knows at a glance that it is fixed. In Python, a variable assigned directly in the class body, `KILOGRAMS_PER_POUND = 0.45359237`, plays the same role; capitals again signal “do not change”.
- `public Q15Weight(double p) { pounds = p; }` — Java converts the `int` 150 in `new Q15Weight(150)` to the `double` 150.0 automatically, a *widening conversion* that never loses information.

- `return pounds * KILOGRAMS_PER_POUND;` — the kilograms are recomputed at every call, which costs one multiplication, nothing.
- `Math.abs(expected - actual) < 1e-9` — `1e-9` is scientific notation for 10^{-9}. Comparing two computed `double` values with `==` is almost always a mistake; comparing their distance with a small tolerance is the standard remedy. Python's `assertAlmostEqual(a, b, places=9)` rounds the difference to nine decimal places and checks that it is zero.

4.7.7 A tempting wrong solution

Listing 4.24 stores both units and adds a setter that forgets one of them.

Listing 4.24 `WrongWeight.java`: two fields that can fall out of step.

```java
/** Storing kilograms as a second field lets it fall out of step with pounds. */
public class WrongWeight {
    private double pounds;
    private double kilograms;

    public WrongWeight(double p) {
        pounds = p;
        kilograms = p * 0.45359237;
    }

    public void setPounds(double p) {
        pounds = p; // forgets to update kilograms
    }

    public static void main(String[] args) {
        WrongWeight w = new WrongWeight(150);
        w.setPounds(140);
        System.out.println(w.pounds + " lb = " + w.kilograms + " kg ?");
    }
}
```

```
140.0 lb = 68.0388555 kg ?
```

The object now claims that 140 pounds are 68.04 kilograms, which is the value for 150 pounds. No compiler warns about this; only a test that sets and then reads catches it. **Listing 4.25** makes the same mistake in Python.

Listing 4.25 `wrong_weight.py`: the same stale second value in Python.

```python
"""Storing kilograms as a second attribute lets it fall out of step with pounds."""


class WrongWeight:
    def __init__(self, p):
        self.pounds = p
        self.kilograms = p * 0.45359237

    def set_pounds(self, p):
        self.pounds = p  # forgets to update kilograms


w = WrongWeight(150.0)
w.set_pounds(140.0)
print(f"{w.pounds} lb = {w.kilograms} kg ?")
```

```
140.0 lb = 68.0388555 kg ?
```

The general rule is worth remembering: *store each fact once, and compute everything that follows from it.*

CHECK YOURSELF. Add a method `getStonesAndPounds()` that returns a string such as `"11 st 6.0 lb"` for a weight of 160 pounds (one stone is 14 pounds). What does it return for 150 pounds?

Answer. Whole stones are `(int) (pounds / 14)`, and the rest is `pounds - 14 * stones`; in Python `int(self._pounds // 14)` and the same subtraction. For 150 pounds: $150/14 = 10.71\ldots$, so 10 stones, and $150 - 140 = 10$ pounds: `"10 st 10.0 lb"`.

4.8 Problem 4.6 — A Fraction Class

4.8.1 The problem

PROBLEM 4.6. Problem 4.1 added and multiplied two fractions given as loose numbers. Now represent fractions by a Java class `Q16Fraction`. Besides the constructor and the accessor methods `getNumerator` and `getDenominator`, the class has a method `toString` (to print a fraction) and methods `getSum` and `getProduct`, which produce the sum and the product of two fractions and return them as a `Q16Fraction`. For instance, after `Q16Fraction f1 = new Q16Fraction(1, 2);` and `Q16Fraction f2 = new Q16Fraction(3, 7);`, `f1.toString()` returns the string `"1/2"`, `System.out.print(f2.getProduct(f1));` prints `3/14`, and `System.out.print(f2.getSum(f1));` prints `13/14`. Test the class in `Q16FractionTest`.

4.8.2 Understanding it by hand

This is Problem 4.1 again, but each fraction is now *one thing*, an object that knows its own top and bottom. The formulas are still Equation (4.1) and Equation (4.2). With $f_2 = \frac{3}{7}$ and $f_1 = \frac{1}{2}$:

- `f2.getProduct(f1)`: $\frac{3 \cdot 1}{7 \cdot 2} = \frac{3}{14}$;
- `f2.getSum(f1)`: $\frac{3 \cdot 2 + 7 \cdot 1}{7 \cdot 2} = \frac{6+7}{14} = \frac{13}{14}$;
- after both calls, f_2 must still be $\frac{3}{7}$: asking for a sum must not change the fraction you asked.

Because the results are fractions themselves, they can be used again at once — this is the real gain over Problem 4.1. For a test of that, chain two operations: $(f_1 + f_1) \cdot (f_2 + f_2)$. By hand, $f_1 + f_1 = \frac{1 \cdot 2 + 2 \cdot 1}{2 \cdot 2} = \frac{4}{4}$ and $f_2 + f_2 = \frac{3 \cdot 7 + 7 \cdot 3}{7 \cdot 7} = \frac{42}{49}$, and their product is $\frac{4 \cdot 42}{4 \cdot 49} = \frac{168}{196}$. Unreduced fractions grow quickly; $\frac{168}{196}$ is just $\frac{6}{7}$.

4.8.3 Classes, fields and methods

```
+-------------------------------------------------+
| Q16Fraction                                     |
+-------------------------------------------------+
| - numerator : int                               |
| - denominator : int                             |
+-------------------------------------------------+
| + Q16Fraction(numerator, denominator)           |
| + getNumerator() : int                          |
| + getDenominator() : int                        |
| + toString() : String                           |
| + getProduct(other : Q16Fraction) : Q16Fraction |
| + getSum(other : Q16Fraction) : Q16Fraction     |
+-------------------------------------------------+
```

Figure 4.4 shows, in its lower box, what happens in memory: `f1` and `f2` refer to two objects, and `f2.getSum(f1)` creates a *third* object for the sum. An object that never changes after it is built is

called *immutable*; immutable objects are easier to reason about and to share, which is why Java's own `String` is immutable [8].

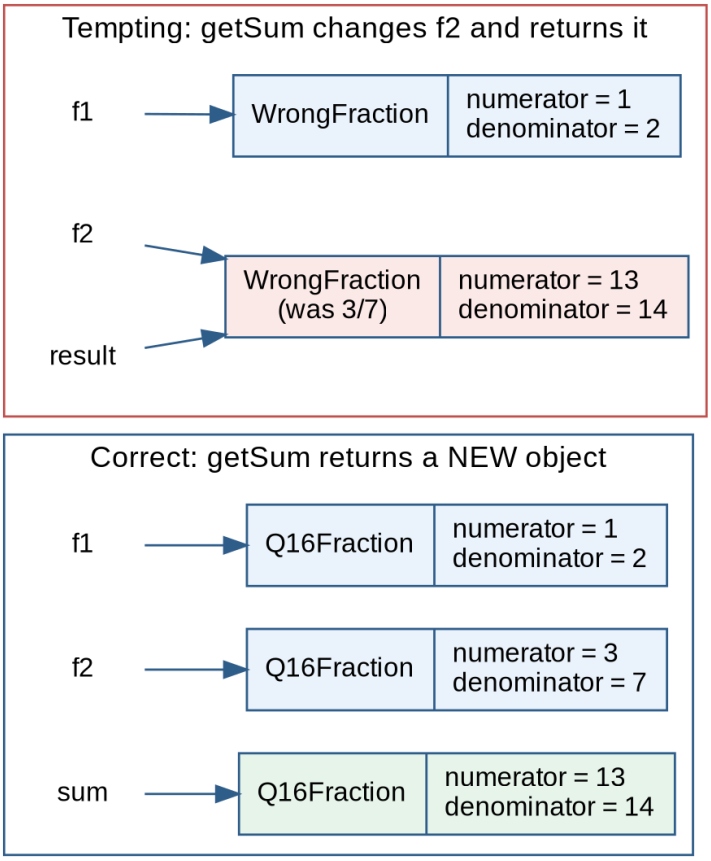


Figure 4.4 Lower box: the correct `getSum` leaves `f1` and `f2` alone and returns a new object. Upper box: the tempting version of Section 4.8.7 overwrites `f2` and returns it, so `f2` and the result are the same changed object.

4.8.4 The Java solution

Listing 4.26 is the class and **Listing 4.27** the driver, which prints the three required outputs before its checks.

Listing 4.26 `Q16Fraction.java`: an immutable fraction whose operations return new fractions.

```java
/**
 * Problem 4.6: a fraction numerator/denominator as an object.
 * getSum and getProduct build and return a NEW fraction; neither
 * operand is changed. Results are not reduced to lowest terms.
 */
public class Q16Fraction {
    private int numerator;
    private int denominator;

    public Q16Fraction(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public int getNumerator() {
        return numerator;
    }

    public int getDenominator() {
        return denominator;
    }

    @Override
    public String toString() {
        return numerator + "/" + denominator;
    }

    public Q16Fraction getProduct(Q16Fraction other) {
        return new Q16Fraction(numerator * other.getNumerator(),
                denominator * other.getDenominator());
    }

    public Q16Fraction getSum(Q16Fraction other) {
        return new Q16Fraction(numerator * other.getDenominator() + denominator *
other.getNumerator(),
                denominator * other.getDenominator());
    }
}
```

Listing 4.27 `Q16FractionTest.java`: the three required outputs, immutability, and a chained computation.

```java
/** Plain test driver for Q16Fraction: run it with "java Q16FractionTest". */
public class Q16FractionTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        Q16Fraction f1 = new Q16Fraction(1, 2);
        Q16Fraction f2 = new Q16Fraction(3, 7);
        System.out.println(f1);
        System.out.println(f2.getProduct(f1));
        System.out.println(f2.getSum(f1));

        check("toString", "1/2", f1.toString());
        check("product", "3/14", f2.getProduct(f1).toString());
        check("sum", "13/14", f2.getSum(f1).toString());
        check("f2 unchanged", "3/7", f2.toString());
        check("chained", "168/196", f1.getSum(f1).getProduct(f2.getSum(f2)).toString());
        check("getNumerator", 3, f2.getNumerator());
        check("getDenominator", 7, f2.getDenominator());
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

```
1/2
3/14
13/14
PASS toString
PASS product
PASS sum
PASS f2 unchanged
PASS chained
PASS getNumerator
PASS getDenominator
7 passed, 0 failed
```

The first three lines are exactly the outputs the problem demands: `System.out.println(f1)` calls `toString` automatically.

4.8.5 The Python solution

Listing 4.28 is a line-for-line translation.

Listing 4.28 `q16_fraction.py`: Problem 4.6 in Python.

```
"""Problem 4.6: a fraction numerator/denominator as an object.
get_sum and get_product build and return a NEW fraction; neither
operand is changed. Results are not reduced to lowest terms.
"""
import unittest


class Q16Fraction:
    def __init__(self, numerator, denominator):
        self._numerator = numerator
        self._denominator = denominator

    def get_numerator(self):
        return self._numerator

    def get_denominator(self):
        return self._denominator

    def __str__(self):
        return f"{self._numerator}/{self._denominator}"

    def get_product(self, other):
        return Q16Fraction(self._numerator * other.get_numerator(),
                           self._denominator * other.get_denominator())

    def get_sum(self, other):
        return Q16Fraction(self._numerator * other.get_denominator()
                           + self._denominator * other.get_numerator(),
                           self._denominator * other.get_denominator())


class TestQ16Fraction(unittest.TestCase):
    def setUp(self):
        self.f1 = Q16Fraction(1, 2)
        self.f2 = Q16Fraction(3, 7)

    def test_str(self):
        self.assertEqual(str(self.f1), "1/2")

    def test_product_and_sum(self):
        self.assertEqual(str(self.f2.get_product(self.f1)), "3/14")
        self.assertEqual(str(self.f2.get_sum(self.f1)), "13/14")
```

Listing 4.28 (Continued) *q16_fraction.py: Problem 4.6 in Python.*

```
    def test_operands_unchanged(self):
        self.f2.get_sum(self.f1)
        self.assertEqual(str(self.f2), "3/7")

    def test_chained(self):
        result = self.f1.get_sum(self.f1).get_product(self.f2.get_sum(self.f2))
        self.assertEqual(str(result), "168/196")


if __name__ == "__main__":
    f1 = Q16Fraction(1, 2)
    f2 = Q16Fraction(3, 7)
    print(f1)
    print(f2.get_product(f1))
    print(f2.get_sum(f1))
    unittest.main(verbosity=2)
```

```
1/2
3/14
13/14
test_chained (__main__.TestQ16Fraction.test_chained) ... ok
test_operands_unchanged (__main__.TestQ16Fraction.test_operands_unchanged) ... ok
test_product_and_sum (__main__.TestQ16Fraction.test_product_and_sum) ... ok
test_str (__main__.TestQ16Fraction.test_str) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

4.8.6 Line by line

- `public Q16Fraction getProduct(Q16Fraction other)` — the parameter type and the return type are both our own class. A class is a type like `int`, and can be used anywhere a type can.
- `return new Q16Fraction(numerator * other.getNumerator(), denominator * other.getDenominator());` — this reads as Equation (4.2): `numerator` and `denominator` are this object's fields, and `other.getNumerator()` asks the other object for its top.
- The fields have no setters. Problem 4.6 does not ask for any, and leaving them out is what makes the class immutable.
- `f1.getSum(f1).getProduct(f2.getSum(f2))` — Java evaluates this from the inside out and left to right: first `f1.getSum(f1)`, which is an object, then `f2.getSum(f2)`, another object, then `getProduct` of the first with the second. *Chaining* like this only works because each method returns an object.
- In Python, `str(self.f2.get_sum(self.f1))` calls `__str__` on the returned object, which is how the tests compare the text `"13/14"`.

4.8.7 A tempting wrong solution

The tempting version computes the sum *in place*: it overwrites the fields of `this` and returns `this`. It prints the right answer the first time, which is exactly what makes it dangerous (**Listing 4.29**).

Listing 4.29 `WrongFraction.java`: a `getSum` that changes the object it is called on.

```java
/** A getSum that changes the fraction it is called on, instead of making a new one. */
public class WrongFraction {
    private int numerator;
    private int denominator;

    public WrongFraction(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public WrongFraction getSum(WrongFraction other) {
        numerator = numerator * other.denominator + denominator * other.numerator;
        denominator = denominator * other.denominator;
        return this;
    }

    @Override
    public String toString() {
        return numerator + "/" + denominator;
    }

    public static void main(String[] args) {
        WrongFraction f1 = new WrongFraction(1, 2);
        WrongFraction f2 = new WrongFraction(3, 7);
        System.out.println("f2 + f1 = " + f2.getSum(f1));
        System.out.println("f2 is now " + f2);
        System.out.println("f2 + f1 = " + f2.getSum(f1) + " (same call, new answer)");
    }
}
```

```
f2 + f1 = 13/14
f2 is now 13/14
f2 + f1 = 40/28 (same call, new answer)
```

The first line is correct. The second shows that `f2` itself has become $\frac{13}{14}$, as in the upper box of **Figure 4.4**. The third shows the consequence: the very same expression, `f2.getSum(f1)`, now gives $\frac{40}{28}$. A method whose answer depends on how often it has been called is a trap for everyone who uses it. **Listing 4.30** shows that Python behaves the same way.

Listing 4.30 `wrong_fraction.py`: the same in-place sum in Python.

```python
"""A get_sum that changes the fraction it is called on, instead of making a new one."""


class WrongFraction:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def get_sum(self, other):
        self.numerator = self.numerator * other.denominator + self.denominator *
other.numerator
        self.denominator = self.denominator * other.denominator
        return self

    def __str__(self):
        return f"{self.numerator}/{self.denominator}"


f1 = WrongFraction(1, 2)
f2 = WrongFraction(3, 7)
print(f"f2 + f1 = {f2.get_sum(f1)}")
print(f"f2 is now {f2}")
print(f"f2 + f1 = {f2.get_sum(f1)} (same call, new answer)")
```

```
f2 + f1 = 13/14
f2 is now 13/14
f2 + f1 = 40/28 (same call, new answer)
```

CHECK YOURSELF. Make the constructor store every fraction in lowest terms, so that `new Q16Fraction(168, 196)` prints `6/7`. Use the greatest common divisor, computed by Euclid's algorithm with a `while` loop (Problem 4.13 traces it). What do `f2.getSum(f2)` and `new Q16Fraction(2, 6)` print then?

Answer. In the constructor, compute `int g = gcd(Math.abs(numerator), denominator);` with `gcd(a, b)` = `while (b != 0) { int r = a % b; a = b; b = r; } return a;`, and store `numerator / g` and `denominator / g` (if `g` is 0, which happens only for $\frac{0}{0}$, store the numbers unchanged). Then $\frac{42}{49}$ becomes $\frac{6}{7}$ because $\gcd(42,49) = 7$, and $\frac{2}{6}$ becomes $\frac{1}{3}$ because $\gcd(2,6) = 2$. The chained test of **Listing 4.27** must then expect `6/7`.

4.9 Problem 4.7 — Player and Enemy

4.9.1 The problem

PROBLEM 4.7. Define two opposing classes, `Player` and `Enemy`. Both have a name, health, power and defense. Define a constructor, getters and setters for these attributes. Implement in both classes: (a) `attack`, which takes the opposing class as its parameter; the opposing object then calls the method below with this object's attack; (b) `takeDamage`, which takes the opposing attack as its parameter and reduces this object's health by (opposing attack − this defense). If the health falls below zero, it prints `"{name} died!"`. Test both classes in `EnemyPlayerTest`.

4.9.2 Understanding it by hand

Two objects talk to each other. When the hero attacks the villain, the hero does *not* reach into the villain and lower its health; it tells the villain "you have been hit with power 25", and the villain works out the damage from *its own* defense. That division of labor is the object-oriented heart of the problem: each object is responsible for its own fields.

Take a hero with health 100, power 25, defense 10, and a villain with health 80, power 20, defense 5. **Table 4.4** follows the fight.

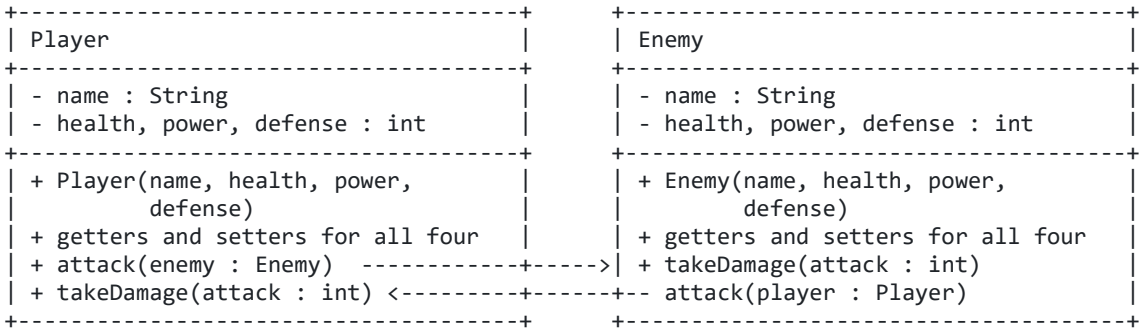
Table 4.4 The fight of Problem 4.7 worked by hand: each attack costs the defender (attacker's power − defender's defense).

Action	Damage	Hero's health	Villain's health
start		100	80
hero attacks villain	$25 - 5 = 20$	100	60
villain attacks hero	$20 - 10 = 10$	90	60
hero attacks villain (round 1)	20	90	40
hero attacks villain (round 2)	20	90	20
hero attacks villain (round 3)	20	90	0 — "Villain died!"

Two questions are left open by the statement. First, what if the defense is larger than the attack — a "tank" with defense 40 hit with power 25? The formula gives damage $25 - 40 = -15$, and "reducing health by -15" would *heal* the tank. No game works that way, so we treat negative

damage as zero. Second, the statement says “below zero”, yet a character left with exactly 0 health, like our villain, is surely dead too. We print the message when health is *zero or below*, and say so in a comment. Reading a specification carefully includes noticing where it is silent or slightly off, and writing your decision down.

4.9.3 Classes, fields and methods



The two classes are mirror images. (Chapter 7 will show how *inheritance* lets them share this code instead of repeating it; Chapter 8 returns to exactly this pair.)

4.9.4 The Java solution

Listing 4.31 is `Player`. The file `Enemy.java` in the code folder is the same class with the words `Player` and `Enemy` exchanged (and `enemy` and `player` in the parameter of `attack`), so we do not print it twice.

Listing 4.31 `Player.java`: one side of the fight; `Enemy.java` is its mirror image.

```java
/**
 * Problem 4.7: one side of a two-sided fight. A Player attacks an Enemy;
 * the Enemy decides for itself how much the attack hurts.
 */
public class Player {
    private String name;
    private int health;
    private int power;
    private int defense;

    public Player(String name, int health, int power, int defense) {
        this.name = name;
        this.health = health;
        this.power = power;
        this.defense = defense;
    }

    public String getName() {
        return name;
    }

    public int getHealth() {
        return health;
    }

    public int getPower() {
        return power;
    }

    public int getDefense() {
        return defense;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setHealth(int health) {
        this.health = health;
    }

    public void setPower(int power) {
        this.power = power;
    }
```

Listing 4.31 (Continued) *Player.java: one side of the fight; Enemy.java is its mirror image.*

```
    public void setDefense(int defense) {
        this.defense = defense;
    }

    public void attack(Enemy enemy) {
        enemy.takeDamage(power);
    }

    public void takeDamage(int attack) {
        int damage = attack - defense;
        if (damage < 0) {
            damage = 0; // a strong defense blocks the blow; it never heals
        }
        health = health - damage;
        if (health <= 0) {
            System.out.println(name + " died!");
        }
    }
}
```

Listing 4.32 replays **Table 4.4**.

Listing 4.32 `EnemyPlayerTest.java`: the fight of **Table 4.4**, with a `while` loop for the last rounds.

```
/** Plain test driver for Player and Enemy: run it with "java EnemyPlayerTest". */
public class EnemyPlayerTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        Player hero = new Player("Hero", 100, 25, 10);
        Enemy villain = new Enemy("Villain", 80, 20, 5);

        hero.attack(villain);
        check("villain loses 25 - 5 = 20", 60, villain.getHealth());
        villain.attack(hero);
        check("hero loses 20 - 10 = 10", 90, hero.getHealth());

        int round = 1;
        while (villain.getHealth() > 0) {
            hero.attack(villain);
            System.out.println("Round " + round + ": " + villain.getName()
                    + " has " + villain.getHealth() + " health left");
            round++;
        }
        check("villain ends at 0", 0, villain.getHealth());

        Enemy tank = new Enemy("Tank", 50, 5, 40);
        hero.attack(tank);
        check("defense 40 blocks attack 25", 50, tank.getHealth());
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

```
PASS villain loses 25 - 5 = 20
PASS hero loses 20 - 10 = 10
Round 1: Villain has 40 health left
Round 2: Villain has 20 health left
Villain died!
Round 3: Villain has 0 health left
PASS villain ends at 0
PASS defense 40 blocks attack 25
4 passed, 0 failed
```

Look at the order of the lines around the villain's death: `Villain died!` is printed *inside* the third `hero.attack(villain)` call, before the loop prints `Round 3`. Output order is a faithful record of call order.

4.9.5 The Python solution

Listing 4.33 keeps both classes in one module.

Listing 4.33 `player_enemy.py`: both classes of Problem 4.7 in one Python module.

```
"""Problem 4.7: two opposing sides of a fight. Each side attacks the other;
the side that is hit decides for itself how much the attack hurts.
"""
import unittest


class Player:
    def __init__(self, name, health, power, defense):
        self._name = name
        self._health = health
        self._power = power
        self._defense = defense

    def get_name(self):
        return self._name

    def get_health(self):
        return self._health

    def get_power(self):
        return self._power

    def get_defense(self):
        return self._defense

    def set_name(self, name):
        self._name = name

    def set_health(self, health):
        self._health = health

    def set_power(self, power):
        self._power = power

    def set_defense(self, defense):
        self._defense = defense

    def attack(self, enemy):
        enemy.take_damage(self._power)
```

Listing 4.33 (Continued) *player_enemy.py: both classes of Problem 4.7 in one Python module.*

```
    def take_damage(self, attack):
        damage = attack - self._defense
        if damage < 0:
            damage = 0  # a strong defense blocks the blow; it never heals
        self._health = self._health - damage
        if self._health <= 0:
            print(f"{self._name} died!")


class Enemy:
    def __init__(self, name, health, power, defense):
        self._name = name
        self._health = health
        self._power = power
        self._defense = defense

    def get_name(self):
        return self._name

    def get_health(self):
        return self._health

    def get_power(self):
        return self._power

    def get_defense(self):
        return self._defense

    def set_name(self, name):
        self._name = name

    def set_health(self, health):
        self._health = health

    def set_power(self, power):
        self._power = power

    def set_defense(self, defense):
        self._defense = defense

    def attack(self, player):
        player.take_damage(self._power)
```

Listing 4.33 (Continued) *player_enemy.py: both classes of Problem 4.7 in one Python module.*

```
    def take_damage(self, attack):
        damage = attack - self._defense
        if damage < 0:
            damage = 0  # a strong defense blocks the blow; it never heals
        self._health = self._health - damage
        if self._health <= 0:
            print(f"{self._name} died!")


class TestPlayerEnemy(unittest.TestCase):
    def test_exchange_of_blows(self):
        hero = Player("Hero", 100, 25, 10)
        villain = Enemy("Villain", 80, 20, 5)
        hero.attack(villain)
        self.assertEqual(villain.get_health(), 60)
        villain.attack(hero)
        self.assertEqual(hero.get_health(), 90)

    def test_defense_blocks(self):
        hero = Player("Hero", 100, 25, 10)
        tank = Enemy("Tank", 50, 5, 40)
        hero.attack(tank)
        self.assertEqual(tank.get_health(), 50)


if __name__ == "__main__":
    hero = Player("Hero", 100, 25, 10)
    villain = Enemy("Villain", 80, 20, 5)
    hero.attack(villain)
    villain.attack(hero)
    round_number = 1
    while villain.get_health() > 0:
        hero.attack(villain)
        print(f"Round {round_number}: {villain.get_name()} has {villain.get_health()}
health left")
        round_number += 1
    unittest.main(verbosity=2)
```

```
Round 1: Villain has 40 health left
Round 2: Villain has 20 health left
Villain died!
Round 3: Villain has 0 health left
test_defense_blocks (__main__.TestPlayerEnemy.test_defense_blocks) ... ok
test_exchange_of_blows (__main__.TestPlayerEnemy.test_exchange_of_blows) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

4.9.6 Line by line

- `public void attack(Enemy enemy) { enemy.takeDamage(power); }` — the whole method is one *message*: “take damage from an attack of this power”. The player never touches the enemy’s fields.
- `int damage = attack - defense; if (damage < 0) { damage = 0; }` — the formula of the statement, then the guard against healing.
- `health = health - damage; if (health <= 0) { … }` — subtract first, then test; testing before subtracting would announce a death one blow late.

- `while (villain.getHealth() > 0)` in the driver — the loop does not know in advance how many blows are needed; it repeats *while* the condition holds, exactly the kind of situation Chapter 3 recommends `while` for over `for`.
- In Python both classes live in one file. Python does not require one class per file; it is simply convenient here because the two classes always travel together.

4.9.7 A tempting wrong solution

It is tempting to let `attack` do all the work: `villain.health = villain.health - power;`. **Listing 4.34** tries even less — it simply sets the enemy's health to zero from outside.

Listing 4.34 `Cheater.java`: code outside `Enemy` tries to write its private field.

```
/** Reaching into another object's private field: the compiler refuses. */
public class Cheater {
    public static void main(String[] args) {
        Enemy villain = new Enemy("Villain", 80, 20, 5);
        villain.health = 0;
        System.out.println(villain.getHealth());
    }
}
```

Compiled with `javac -cp .. Cheater.java` from the `tempting` folder (the `-cp ..` tells the compiler where to find `Enemy`), it does not even compile:

```
Cheater.java:5: error: health has private access in Enemy
        villain.health = 0;
               ^
1 error
```

This error is a *good* thing: the compiler enforces the rule that only `Enemy` changes an `Enemy`'s health. If it did not, every class in the program could bypass the defense rule and the death message. Python, as **Listing 4.35** shows, relies on the underscore convention only, and lets the cheater through without a word.

Listing 4.35 `cheater.py`: Python does not enforce privacy, so the rule lives in the programmer's discipline.

```
"""Reaching into another object's 'private' attribute: Python does not stop you."""
from player_enemy import Enemy

villain = Enemy("Villain", 80, 20, 5)
villain._health = 0
print(villain.get_health())
```

```
0
```

The villain is now at 0 health, yet it never printed "Villain died!", because the one method that knows about death was bypassed. In Python the underscore is a promise between programmers, and keeping it is your responsibility.

> **CHECK YOURSELF.** Add a method `isAlive()` to both classes, and change `takeDamage` so that a character who is already dead ignores further attacks (and does not print "died!" a second time). After the fight of **Table 4.4**, what does one more `hero.attack(villain)` print, and what is the villain's health?
>
> *Answer.* `public boolean isAlive() { return health > 0; }`, and `takeDamage` begins with `if (!isAlive()) { return; }`. The extra attack prints nothing, and the villain's health stays 0. Without the guard the health would become −20 and "Villain died!" would appear again.

4.10 Problem 4.8 — Money in Rupiah

The remaining seven problems are new. They use nothing beyond Chapters 1 to 3, and they come from the kind of small business program that Profitina Laundry, the case study of this book, needs every day.

4.10.1 The problem

> **PROBLEM 4.8.** Write a class `Rupiah` for an amount of money in whole rupiah. It has a constructor taking the amount, a getter `getAmount`, and three methods that return new `Rupiah` objects: `plus(other)` for a sum, `times(quantity)` for a price times a whole quantity, and `discount(percent)` for the price after a discount of the given whole percent, rounded down to a whole rupiah. Its `toString` writes the amount the Indonesian way, with a dot between every group of three digits counted from the right: `Rp1.250.000`, `Rp999`, `-Rp5.000`.

4.10.2 Understanding it by hand

Money must be exact. A laundry that adds up a thousand receipts cannot tolerate a total that is off by a fraction of a rupiah, so we store the amount as a whole number of type `long` (a 64-bit integer, large enough for 9.2×10^{18} rupiah) and never as a `double`.

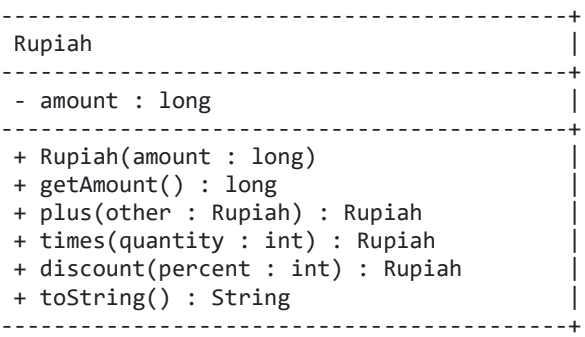
The order that Problem 4.10 will need: four kilograms of washing at Rp7.000 plus one item at Rp12.500 is $4 \cdot 7000 + 12500 = 28000 + 12500 = 40500$. Ten percent off is $40500 \cdot \frac{90}{100} = \frac{3645000}{100} = 36450$. An awkward discount: 15 percent off Rp9.999 is $\frac{9999 \cdot 85}{100} = \frac{849915}{100} = 8499.15$, rounded down to 8499. Multiplying *before* dividing keeps the arithmetic whole until the very last step.

The format is the real string-processing exercise. Take the digits `1250000`. Walk from the *right*, copying one digit at a time to the front of the result, and after every third digit put a dot — unless that digit was the leftmost one:

Table 4.5 Grouping the digits of 1250000 from the right: a dot is added after every third digit, except after the leftmost one.

Digit read (from the right)	Count	Result so far
`0`	1	`0`
`0`	2	`00`
`0`	3	`.000`
`0`	4	`0.000`
`5`	5	`50.000`
`2`	6	`.250.000`
`1`	7	`1.250.000`

4.10.3 Classes, fields and methods



Like `Q16Fraction`, `Rupiah` is immutable: every operation returns a new object.

4.10.4 The Java solution

Listing 4.36 is the class; its `toString` is the procedure of **Table 4.5** written as a loop. **Listing 4.37** tests the group boundaries 999 and 1000.

Listing 4.36 `Rupiah.java`: exact money in whole rupiah, formatted with dots from the right.

```
/**
 * Problem 4.8: an amount of money in whole rupiah. The amount is a long,
 * never a double, so sums are exact. toString groups the digits in threes
 * with dots, the Indonesian way: 1250000 becomes "Rp1.250.000".
 */
public class Rupiah {
    private long amount;

    public Rupiah(long amount) {
        this.amount = amount;
    }

    public long getAmount() {
        return amount;
    }

    public Rupiah plus(Rupiah other) {
        return new Rupiah(amount + other.amount);
    }

    public Rupiah times(int quantity) {
        return new Rupiah(amount * quantity);
    }

    // Price after a discount of the given percent, rounded down to a whole rupiah.
    public Rupiah discount(int percent) {
        return new Rupiah(amount * (100 - percent) / 100);
    }

    @Override
    public String toString() {
        String digits = Long.toString(Math.abs(amount));
        String grouped = "";
        int count = 0;
        for (int i = digits.length() - 1; i >= 0; i--) { // walk from the RIGHT
            grouped = digits.charAt(i) + grouped;
            count++;
            if (count % 3 == 0 && i > 0) {
                grouped = "." + grouped;
            }
        }
        if (amount < 0) {
            return "-Rp" + grouped;
        }
        return "Rp" + grouped;
    }
```

Listing 4.36 (Continued) *Rupiah.java: exact money in whole rupiah, formatted with dots from the right.*

```
}
```

Listing 4.37 `RupiahTest.java`: formatting at the group boundaries, and exact arithmetic.

```java
/** Plain test driver for Rupiah: run it with "java RupiahTest". */
public class RupiahTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        Rupiah wash = new Rupiah(7000);
        Rupiah order = wash.times(4).plus(new Rupiah(12500));
        System.out.println("Order: " + order);
        System.out.println("After 10% off: " + order.discount(10));

        check("zero", "Rp0", new Rupiah(0).toString());
        check("three digits", "Rp999", new Rupiah(999).toString());
        check("four digits", "Rp1.000", new Rupiah(1000).toString());
        check("seven digits", "Rp1.250.000", new Rupiah(1250000).toString());
        check("negative", "-Rp5.000", new Rupiah(-5000).toString());
        check("sum", 40500L, order.getAmount());
        check("discount rounds down", 36450L, order.discount(10).getAmount());
        check("8499.15 rounds down", 8499L, new Rupiah(9999).discount(15).getAmount());
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

```
Order: Rp40.500
After 10% off: Rp36.450
PASS zero
PASS three digits
PASS four digits
PASS seven digits
PASS negative
PASS sum
PASS discount rounds down
PASS 8499.15 rounds down
8 passed, 0 failed
```

4.10.5 The Python solution

Listing 4.38 follows the same loop.

Listing 4.38 `rupiah.py`: Problem 4.8 in Python.

```
"""Problem 4.8: an amount of money in whole rupiah. The amount is an int,
never a float, so sums are exact. str() groups the digits in threes
with dots, the Indonesian way: 1250000 becomes "Rp1.250.000".
"""
import unittest


class Rupiah:
    def __init__(self, amount):
        self._amount = amount

    def get_amount(self):
        return self._amount

    def plus(self, other):
        return Rupiah(self._amount + other._amount)

    def times(self, quantity):
        return Rupiah(self._amount * quantity)

    def discount(self, percent):
        # Price after a discount of the given percent, rounded down to a whole rupiah.
        return Rupiah(self._amount * (100 - percent) // 100)

    def __str__(self):
        digits = str(abs(self._amount))
        grouped = ""
        count = 0
        for i in range(len(digits) - 1, -1, -1):  # walk from the RIGHT
            grouped = digits[i] + grouped
            count += 1
            if count % 3 == 0 and i > 0:
                grouped = "." + grouped
        if self._amount < 0:
            return "-Rp" + grouped
        return "Rp" + grouped


class TestRupiah(unittest.TestCase):
    def test_format(self):
        self.assertEqual(str(Rupiah(0)), "Rp0")
        self.assertEqual(str(Rupiah(999)), "Rp999")
        self.assertEqual(str(Rupiah(1000)), "Rp1.000")
        self.assertEqual(str(Rupiah(1250000)), "Rp1.250.000")
        self.assertEqual(str(Rupiah(-5000)), "-Rp5.000")
```

Listing 4.38 (Continued) *rupiah.py: Problem 4.8 in Python.*

```
    def test_arithmetic(self):
        order = Rupiah(7000).times(4).plus(Rupiah(12500))
        self.assertEqual(order.get_amount(), 40500)
        self.assertEqual(order.discount(10).get_amount(), 36450)
        self.assertEqual(Rupiah(9999).discount(15).get_amount(), 8499)


if __name__ == "__main__":
    order = Rupiah(7000).times(4).plus(Rupiah(12500))
    print(f"Order: {order}")
    print(f"After 10% off: {order.discount(10)}")
    unittest.main(verbosity=2)
```

```
Order: Rp40.500
After 10% off: Rp36.450
test_arithmetic (__main__.TestRupiah.test_arithmetic) ... ok
test_format (__main__.TestRupiah.test_format) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

4.10.6 Line by line

- `String digits = Long.toString(Math.abs(amount));` — turn the absolute value into its digits; the sign is handled separately at the end.
- `for (int i = digits.length() - 1; i >= 0; i--)` — a `for` loop that counts *down*, from the index of the last character to 0. `digits.charAt(i)` is the character at position `i` (the first character has position 0).
- `grouped = digits.charAt(i) + grouped;` — adding a `char` to a `String` puts the character in front. Building a string by repeated `+` creates a new string each time; for a dozen digits that costs nothing, and for long texts the standard class `StringBuilder` does the same job without the copying.
- `if (count % 3 == 0 && i > 0)` — `count % 3 == 0` is "count is a multiple of 3"; `i > 0` is "there are more digits to the left". Both must hold, which `&&` expresses. Without `i > 0`, `999` would become `.999`.
- `amount * (100 - percent) / 100` — Java evaluates left to right: first the multiplication, then the division, which drops the fraction. Python writes the same as `// 100`. For negative amounts the two languages differ (Java's `/` cuts towards zero, Python's `//` rounds down), but a discount is always applied to a positive price.
- Python's `range(len(digits) - 1, -1, -1)` produces the indices from the last down to 0: the stop value `-1` is not included, and the step `-1` counts down.

4.10.7 A tempting wrong solution

Listing 4.39 walks through the digits in the natural reading direction, from the left.

Listing 4.39 `WrongRupiahFormat.java`: grouping from the left, and money in a `double`.

```
/** Grouping digits from the left, and keeping money in a double. */
public class WrongRupiahFormat {
    public static void main(String[] args) {
        String digits = "12500";
        String grouped = "";
        for (int i = 0; i < digits.length(); i++) {
            if (i > 0 && i % 3 == 0) {
                grouped = grouped + ".";
            }
            grouped = grouped + digits.charAt(i);
        }
        System.out.println("from the left: Rp" + grouped);
        System.out.println("double money:  " + 7000 * 1.1);
    }
}
```

```
from the left: Rp125.00
double money:  7700.000000000001
```

Counting from the left puts the dot after `125`, turning twelve thousand five hundred into something that looks like one hundred twenty-five. Groups of three belong to the *thousands*, *millions* and so on, which are counted from the right. The second line shows why money is never a `double`: adding 10 percent to Rp7.000 by multiplying with `1.1` gives `7700.000000000001`, because 1.1 has no exact binary form. **Listing 4.40** shows the same two results in Python.

Listing 4.40 `wrong_rupiah_format.py`: the same two mistakes in Python.

```
"""Grouping digits from the left, and keeping money in a float."""
digits = "12500"
grouped = ""
for i in range(len(digits)):
    if i > 0 and i % 3 == 0:
        grouped = grouped + "."
    grouped = grouped + digits[i]
print(f"from the left: Rp{grouped}")
print(f"float money:   {7000 * 1.1}")
```

```
from the left: Rp125.00
float money:   7700.000000000001
```

CHECK YOURSELF. Add a method `split(int people)` that returns the share of each person when a bill is divided evenly, rounded *down*, and a method `remainder(int people)` that returns what is left over. For Rp100.000 and 3 people, what do they return, and why is the remainder needed?

Answer. `split` returns `new Rupiah(amount / people)` and `remainder` returns `new Rupiah(amount % people)`: Rp33.333 each, and Rp1 left over. Without the remainder, $3 \cdot 33333 = 99999$ and one rupiah would silently vanish; with it, the shares plus the remainder add up to the bill exactly.

4.11 Problem 4.9 — Time of Day

4.11.1 The problem

PROBLEM 4.9. Profitina Laundry promises that a service is ready a certain number of minutes after the customer drops it off. Write a class `TimeOfDay` for a time on a 24-hour clock with a constructor `TimeOfDay(hours, minutes)`, getters `getHours` and `getMinutes`, a method `plusMinutes(m)` that returns the time `m` minutes later (or earlier, for negative `m`), wrapping around midnight, a method `isBefore(other)`, a method `minutesUntil(other)` that returns how many minutes lie from this time forward to the other, and a `toString` that always prints two digits for the hour and two for the minutes, as in `08:05`.

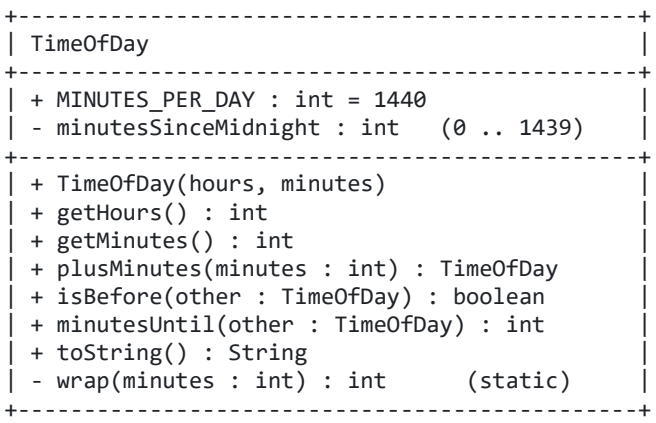
4.11.2 Understanding it by hand

Three examples. Dropped off at 22:30, ready 45 minutes later: $30 + 45 = 75$ minutes, which is 1 hour and 15 minutes, so $22 + 1 = 23$ and the time is 23:15 — the minutes *carry* into the hour. Ready 150 minutes after 22:30: 150 minutes is 2 hours 30 minutes, so $22{:}30 + 2{:}30 = 24{:}60$, which is 25:00, which on a 24-hour clock is 01:00 the next morning — the hours *wrap* past midnight. And 20 minutes before 00:10 is 23:50 the previous evening.

The carrying and the wrapping become trivial with one idea: store not two numbers but one, the *minutes since midnight*, from 0 (00:00) to 1439 (23:59). Then 22:30 is $22 \cdot 60 + 30 = 1350$; adding 150 gives 1500; the day has $24 \cdot 60 = 1440$ minutes, and $1500 \bmod 1440 = 60$, which is $60/60 = 1$ hour and $60 \bmod 60 = 0$ minutes: 01:00. Going back 20 minutes from 00:10 gives

$10 - 20 = -10$, and -10 must wrap to 1430, which is 23:50 — this is the negative remainder of Problem 4.2 again.

4.11.3 Classes, fields and methods



The outside world still speaks in hours and minutes — the constructor takes them, and the getters return them — while the inside uses the representation that makes the arithmetic easy. Choosing a good *internal representation*, and hiding it, is what encapsulation buys.

4.11.4 The Java solution

Listing 4.41 is the class and **Listing 4.42** its driver, whose first line is the promise printed on a laundry ticket.

Listing 4.41 `TimeOfDay.java`: a time of day stored as minutes since midnight.

```java
/**
 * Problem 4.9: a time of day on a 24-hour clock, stored as the number of
 * minutes since midnight (0 to 1439). Adding minutes wraps past midnight.
 */
public class TimeOfDay {
    public static final int MINUTES_PER_DAY = 24 * 60;

    private int minutesSinceMidnight;

    public TimeOfDay(int hours, int minutes) {
        minutesSinceMidnight = wrap(hours * 60 + minutes);
    }

    private static int wrap(int minutes) {
        return (minutes % MINUTES_PER_DAY + MINUTES_PER_DAY) % MINUTES_PER_DAY;
    }

    public int getHours() {
        return minutesSinceMidnight / 60;
    }

    public int getMinutes() {
        return minutesSinceMidnight % 60;
    }

    public TimeOfDay plusMinutes(int minutes) {
        return new TimeOfDay(0, minutesSinceMidnight + minutes);
    }

    public boolean isBefore(TimeOfDay other) {
        return minutesSinceMidnight < other.minutesSinceMidnight;
    }

    // Minutes from this time forward to the other time, possibly across midnight.
    public int minutesUntil(TimeOfDay other) {
        return wrap(other.minutesSinceMidnight - minutesSinceMidnight);
    }

    @Override
    public String toString() {
        return String.format("%02d:%02d", getHours(), getMinutes());
    }
}
```

Listing 4.42 `TimeOfDayTest.java`: carry, wrap-around in both directions, padding and comparison.

```java
/** Plain test driver for TimeOfDay: run it with "java TimeOfDayTest". */
public class TimeOfDayTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        TimeOfDay dropOff = new TimeOfDay(22, 30);
        TimeOfDay ready = dropOff.plusMinutes(150);
        System.out.println("Dropped off at " + dropOff + ", ready at " + ready);

        check("zero padding", "08:05", new TimeOfDay(8, 5).toString());
        check("carry into the hour", "23:15", dropOff.plusMinutes(45).toString());
        check("wrap past midnight", "01:00", ready.toString());
        check("going back", "23:50", new TimeOfDay(0, 10).plusMinutes(-20).toString());
        check("one whole day", "22:30", dropOff.plusMinutes(1440).toString());
        check("isBefore", true, new TimeOfDay(9, 0).isBefore(new TimeOfDay(9, 1)));
        check("minutesUntil across midnight", 150, dropOff.minutesUntil(ready));
        check("getHours", 1, ready.getHours());
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

```
Dropped off at 22:30, ready at 01:00
PASS zero padding
PASS carry into the hour
PASS wrap past midnight
PASS going back
PASS one whole day
PASS isBefore
PASS minutesUntil across midnight
PASS getHours
8 passed, 0 failed
```

4.11.5 The Python solution

Listing 4.43 needs no helper for wrapping, because Python's `%` never returns a negative number here.

Listing 4.43 `time_of_day.py`: Problem 4.9 in Python.

```
"""Problem 4.9: a time of day on a 24-hour clock, stored as the number of
minutes since midnight (0 to 1439). Adding minutes wraps past midnight.
"""
import unittest

MINUTES_PER_DAY = 24 * 60


class TimeOfDay:
    def __init__(self, hours, minutes):
        # Python's % with a positive divisor never returns a negative number.
        self._minutes_since_midnight = (hours * 60 + minutes) % MINUTES_PER_DAY

    def get_hours(self):
        return self._minutes_since_midnight // 60

    def get_minutes(self):
        return self._minutes_since_midnight % 60

    def plus_minutes(self, minutes):
        return TimeOfDay(0, self._minutes_since_midnight + minutes)

    def is_before(self, other):
        return self._minutes_since_midnight < other._minutes_since_midnight

    def minutes_until(self, other):
        # Minutes from this time forward to the other time, possibly across midnight.
        return (other._minutes_since_midnight - self._minutes_since_midnight) %
MINUTES_PER_DAY

    def __str__(self):
        return f"{self.get_hours():02d}:{self.get_minutes():02d}"


class TestTimeOfDay(unittest.TestCase):
    def test_format_and_carry(self):
        drop_off = TimeOfDay(22, 30)
        self.assertEqual(str(TimeOfDay(8, 5)), "08:05")
        self.assertEqual(str(drop_off.plus_minutes(45)), "23:15")
        self.assertEqual(str(drop_off.plus_minutes(150)), "01:00")
        self.assertEqual(str(drop_off.plus_minutes(1440)), "22:30")

    def test_going_back(self):
        self.assertEqual(str(TimeOfDay(0, 10).plus_minutes(-20)), "23:50")
```

Listing 4.43 (Continued) *time_of_day.py: Problem 4.9 in Python.*

```
    def test_comparisons(self):
        drop_off = TimeOfDay(22, 30)
        ready = drop_off.plus_minutes(150)
        self.assertTrue(TimeOfDay(9, 0).is_before(TimeOfDay(9, 1)))
        self.assertEqual(drop_off.minutes_until(ready), 150)
        self.assertEqual(ready.get_hours(), 1)


if __name__ == "__main__":
    drop_off = TimeOfDay(22, 30)
    ready = drop_off.plus_minutes(150)
    print(f"Dropped off at {drop_off}, ready at {ready}")
    unittest.main(verbosity=2)
```

```
Dropped off at 22:30, ready at 01:00
test_comparisons (__main__.TestTimeOfDay.test_comparisons) ... ok
test_format_and_carry (__main__.TestTimeOfDay.test_format_and_carry) ... ok
test_going_back (__main__.TestTimeOfDay.test_going_back) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

4.11.6 Line by line

- `minutesSinceMidnight = wrap(hours * 60 + minutes);` — the constructor accepts any numbers and normalizes them, so `new TimeOfDay(0, 1500)` is a legal way to write 01:00. `plusMinutes` uses exactly this: `new TimeOfDay(0, minutesSinceMidnight + minutes)`.
- `private static int wrap(int minutes)` — a *private* helper: the class needs it, but it is not part of what the class offers. The expression `(minutes % 1440 + 1440) % 1440` is the Java idiom of Problem 4.2 for a remainder that is never negative. In Python a single `% MINUTES_PER_DAY` suffices.
- `getHours()` is `minutesSinceMidnight / 60` (integer division, *wanted* here) and `getMinutes()` is `minutesSinceMidnight % 60`.
- `String.format("%02d:%02d", getHours(), getMinutes())` — `%d` prints an integer, `2` asks for at least two characters, and `0` fills with zeros instead of spaces. The Python f-string `f"{h:02d}"` uses the same little language.
- `minutesUntil` wraps the difference, so from 22:30 to 01:00 it answers 150, not $60 - 1350 = -1290$.

4.11.7 A tempting wrong solution

Listing 4.44 `WrongTime.java`: adding minutes without carry, without wrap-around, and without padding.

```java
/** Adding minutes without carry, without wrap-around, and without padding. */
public class WrongTime {
    public static void main(String[] args) {
        int hours = 22, minutes = 30;
        System.out.println("no carry:   " + hours + ":" + (minutes + 45));
        int total = hours * 60 + minutes + 150;
        System.out.println("no wrap:    " + total / 60 + ":" + total % 60);
        System.out.println("no padding: " + 8 + ":" + 5);
    }
}
```

```
no carry:   22:75
no wrap:    25:0
no padding: 8:5
```

Each line is a real bug found in real booking systems: `22:75` (no carry), `25:0` (no wrap and no padding) and `8:5` (no padding). All three disappear once the time is a single number of minutes that is always kept in range. **Listing 4.45** shows the same three results in Python.

Listing 4.45 `wrong_time.py`: the same three slips in Python.

```
"""Adding minutes without carry, without wrap-around, and without padding."""
hours, minutes = 22, 30
print(f"no carry:   {hours}:{minutes + 45}")
total = hours * 60 + minutes + 150
print(f"no wrap:    {total // 60}:{total % 60}")
print(f"no padding: {8}:{5}")
```

```
no carry:   22:75
no wrap:    25:0
no padding: 8:5
```

CHECK YOURSELF. Profitina Laundry is open from 07:00 to 21:00. Write a method `isOpenAt(TimeOfDay t)` (static, or in a small class of your choice). Which of 06:59, 07:00, 20:59 and 21:00 are "open"?

Answer. Open means `!t.isBefore(new TimeOfDay(7, 0)) && t.isBefore(new TimeOfDay(21, 0))`: not before opening time, and before closing time. So 06:59 is closed, 07:00 open, 20:59 open, and 21:00 closed. Writing the two boundaries this way — including the opening minute, excluding the closing minute — is the usual convention for time ranges.

4.12 Problem 4.10 — A Laundry Receipt

4.12.1 The problem

PROBLEM 4.10. Write a class `LaundryReceipt` for Profitina Laundry. A receipt is created for a customer's name and says whether the order is *express*. The method `addService(service, kg)` adds one line: the service is `"wash"` (Rp7.000 per kilogram), `"iron"` (Rp5.000 per kilogram) or `"wash+iron"` (Rp10.000 per kilogram); weight is charged per *started* kilogram, so 3.2 kg is charged as 4 kg; express orders cost 50 percent more. An unknown service or a weight that is not positive is refused: the method returns `false` and adds nothing. The receipt keeps a running total as a `Rupiah` (Problem 4.8), and its `toString` prints the lines and the total in aligned columns.

4.12.2 Understanding it by hand

Sari brings 3.2 kg to be washed and 2.0 kg to be ironed, not express. Washing: 3.2 kg is charged as 4 kg, and $4 \cdot 7000 = 28000$. Ironing: 2.0 kg is exactly 2 started kilograms, and $2 \cdot 5000 = 10000$. Total 38000. Budi brings 4.5 kg to be washed and ironed, express: 5 kg at Rp10.000 is 50000, and half of that again is 25000, so 75000.

"Per started kilogram" is the mathematical *ceiling*: the smallest whole number not less than the weight, written $\lceil 3.2 \rceil = 4$ and $\lceil 2.0 \rceil = 2$. Java's `Math.ceil` and Python's `math.ceil` compute it.

4.12.3 Classes, fields and methods

Figure 4.5 shows the design: a receipt *has* a `Rupiah` total. Using an object of one class as a field of another is called *composition*, and it lets `LaundryReceipt` reuse, rather than repeat, everything Problem 4.8 knows about money.

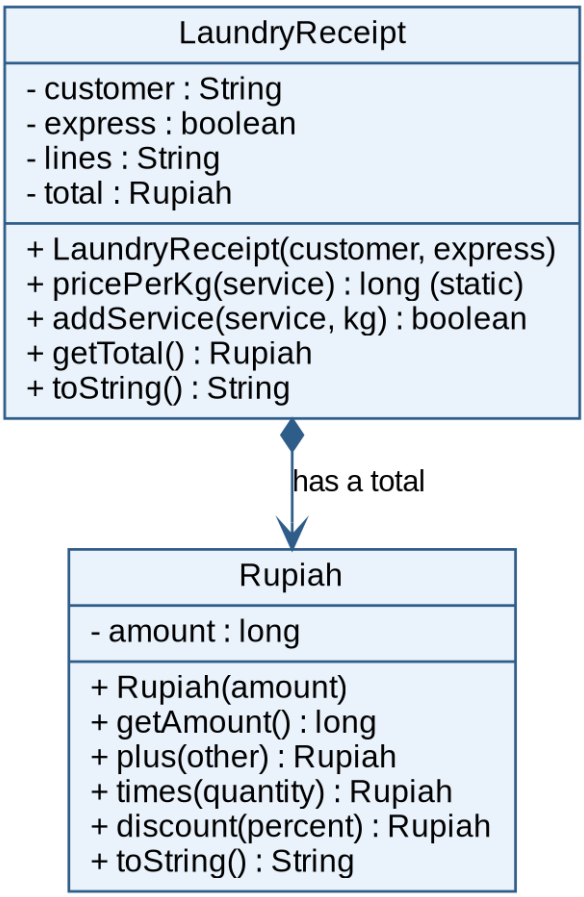


Figure 4.5 The class diagram of Problem 4.10: a `LaundryReceipt` has a `Rupiah` total; the filled diamond marks that the total belongs to the receipt.

4.12.4 The Java solution

Listing 4.46 is the receipt and **Listing 4.47** its driver.

Listing 4.46 `LaundryReceipt.java`: a receipt that prices services with a `switch` and totals them in `Rupiah`.

```java
import java.util.Locale;

/**
 * Problem 4.10: a Profitina Laundry receipt. Each service line is charged
 * per started kilogram (3.2 kg is charged as 4 kg); express service costs
 * 50 percent more. The receipt keeps its lines as text and its total as Rupiah.
 */
public class LaundryReceipt {
    private String customer;
    private boolean express;
    private String lines;
    private Rupiah total;

    public LaundryReceipt(String customer, boolean express) {
        this.customer = customer;
        this.express = express;
        this.lines = "";
        this.total = new Rupiah(0);
    }

    // Price per kilogram of a service, or 0 for a service we do not offer.
    public static long pricePerKg(String service) {
        switch (service) {
            case "wash":
                return 7000;
            case "iron":
                return 5000;
            case "wash+iron":
                return 10000;
            default:
                return 0;
        }
    }

    // Adds one service line; returns false (and adds nothing) for an unknown service.
    public boolean addService(String service, double kg) {
        long perKg = pricePerKg(service);
        if (perKg == 0 || kg <= 0) {
            return false;
        }
        int chargedKg = (int) Math.ceil(kg);
        Rupiah price = new Rupiah(perKg).times(chargedKg);
        if (express) {
            price = price.plus(new Rupiah(price.getAmount() / 2));
        }
        total = total.plus(price);
```

Listing 4.46 (Continued) *LaundryReceipt.java: a receipt that prices services with a switch and totals them in Rupiah.*

```
        // Locale.ROOT prints 3.2, not 3,2, whatever the computer's language is.
        lines = lines + String.format(Locale.ROOT, "%-10s %4.1f kg -> %d kg  %12s%n",
                service, kg, chargedKg, price);
        return true;
    }

    public Rupiah getTotal() {
        return total;
    }

    @Override
    public String toString() {
        String header = "Profitina Laundry - " + customer + (express ? " (express)" : "");
        return String.format(Locale.ROOT, "%s%n%s%-28s%12s", header, lines, "TOTAL",
total);
    }
}
```

Listing 4.47 `LaundryReceiptTest.java`: Sari's and Budi's receipts and the refusals.

```
/** Plain test driver for LaundryReceipt: run it with "java LaundryReceiptTest". */
public class LaundryReceiptTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        LaundryReceipt sari = new LaundryReceipt("Sari", false);
        sari.addService("wash", 3.2);
        sari.addService("iron", 2.0);
        System.out.println(sari);

        LaundryReceipt budi = new LaundryReceipt("Budi", true);
        budi.addService("wash+iron", 4.5);
        System.out.println(budi);

        check("3.2 kg charged as 4 kg", 38000L, sari.getTotal().getAmount());
        check("express adds half", 75000L, budi.getTotal().getAmount());
        check("unknown service refused", false, sari.addService("dry clean", 1.0));
        check("refusal leaves total", 38000L, sari.getTotal().getAmount());
        LaundryReceipt exact = new LaundryReceipt("Test", false);
        exact.addService("iron", 2.0);
        check("exactly 2.0 kg stays 2 kg", 10000L, exact.getTotal().getAmount());
        check("zero weight refused", false, sari.addService("wash", 0));
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

Compile with `javac Rupiah.java LaundryReceipt.java LaundryReceiptTest.java` (or simply `javac *.java`) and run:

```
Profitina Laundry - Sari
wash        3.2 kg -> 4 kg      Rp28.000
iron        2.0 kg -> 2 kg      Rp10.000
TOTAL                           Rp38.000
Profitina Laundry - Budi (express)
wash+iron   4.5 kg -> 5 kg      Rp75.000
TOTAL                           Rp75.000
PASS 3.2 kg charged as 4 kg
PASS express adds half
PASS unknown service refused
PASS refusal leaves total
PASS exactly 2.0 kg stays 2 kg
PASS zero weight refused
6 passed, 0 failed
```

4.12.5 The Python solution

Listing 4.48 prints exactly the same receipts.

Listing 4.48 `laundry_receipt.py`: Problem 4.10 in Python, importing `Rupiah` from `rupiah.py`.

```
"""Problem 4.10: a Profitina Laundry receipt. Each service line is charged
per started kilogram (3.2 kg is charged as 4 kg); express service costs
50 percent more. The receipt keeps its lines as text and its total as Rupiah.
"""
import math
import unittest

from rupiah import Rupiah


def price_per_kg(service):
    # Price per kilogram of a service, or 0 for a service we do not offer.
    match service:
        case "wash":
            return 7000
        case "iron":
            return 5000
        case "wash+iron":
            return 10000
        case _:
            return 0


class LaundryReceipt:
    def __init__(self, customer, express):
        self._customer = customer
        self._express = express
        self._lines = ""
        self._total = Rupiah(0)

    def add_service(self, service, kg):
        # Adds one service line; returns False (and adds nothing) for an unknown service.
        per_kg = price_per_kg(service)
        if per_kg == 0 or kg <= 0:
            return False
        charged_kg = math.ceil(kg)
        price = Rupiah(per_kg).times(charged_kg)
        if self._express:
            price = price.plus(Rupiah(price.get_amount() // 2))
        self._total = self._total.plus(price)
        self._lines += f"{service:<10} {kg:4.1f} kg -> {charged_kg} kg  {str(price):>12}\n"
        return True

    def get_total(self):
        return self._total
```

Listing 4.48 (Continued) *laundry_receipt.py: Problem 4.10 in Python, importing Rupiah from rupiah.py.*

```
    def __str__(self):
        header = f"Profitina Laundry - {self._customer}" + (" (express)" if self._express
else "")
        return f"{header}\n{self._lines}{'TOTAL':<28}{str(self._total):>12}"


class TestLaundryReceipt(unittest.TestCase):
    def test_started_kilograms(self):
        sari = LaundryReceipt("Sari", False)
        sari.add_service("wash", 3.2)
        sari.add_service("iron", 2.0)
        self.assertEqual(sari.get_total().get_amount(), 38000)

    def test_express(self):
        budi = LaundryReceipt("Budi", True)
        budi.add_service("wash+iron", 4.5)
        self.assertEqual(budi.get_total().get_amount(), 75000)

    def test_refusals(self):
        sari = LaundryReceipt("Sari", False)
        self.assertFalse(sari.add_service("dry clean", 1.0))
        self.assertFalse(sari.add_service("wash", 0))
        self.assertEqual(sari.get_total().get_amount(), 0)


if __name__ == "__main__":
    sari = LaundryReceipt("Sari", False)
    sari.add_service("wash", 3.2)
    sari.add_service("iron", 2.0)
    print(sari)
    budi = LaundryReceipt("Budi", True)
    budi.add_service("wash+iron", 4.5)
    print(budi)
    unittest.main(verbosity=2)
```

```
Profitina Laundry - Sari
wash        3.2 kg -> 4 kg      Rp28.000
iron        2.0 kg -> 2 kg      Rp10.000
TOTAL                           Rp38.000
Profitina Laundry - Budi (express)
wash+iron   4.5 kg -> 5 kg      Rp75.000
TOTAL                           Rp75.000
test_express (__main__.TestLaundryReceipt.test_express) ... ok
test_refusals (__main__.TestLaundryReceipt.test_refusals) ... ok
test_started_kilograms (__main__.TestLaundryReceipt.test_started_kilograms) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

4.12.6 Line by line

- `switch (service) { case "wash": return 7000; … default: return 0; }` — a `switch` on a string compares with `equals`, not `==`, so it is safe for typed text. Each `case` returns at once, so no `break` is needed. Python 3.10 and later has the similar `match` statement, where `case _:` is the default.
- `if (perKg == 0 || kg <= 0) { return false; }` — refuse early, before anything is changed. `||` means “or”.
- `int chargedKg = (int) Math.ceil(kg);` — `Math.ceil(3.2)` is the `double` `4.0`, and the cast makes it the `int` 4.

- `price = price.plus(new Rupiah(price.getAmount() / 2));` — the express surcharge, as whole rupiah.
- `String.format(Locale.ROOT, "%-10s %4.1f kg -> %d kg  %12s%n", …)` — `%-10s` is a string left-aligned in 10 characters, `%4.1f` a number with one decimal in 4 characters, `%12s` a string right-aligned in 12, and `%n` a line break. `Locale.ROOT` matters in Indonesia: without it, a computer set to Indonesian prints `3,2` with a decimal *comma* [6]; see below.
- In Python, `f"{service:<10} {kg:4.1f} kg"` uses `<` for left alignment and `>` for right alignment; Python's f-strings ignore the computer's language settings, so they always print `3.2`. `from rupiah import Rupiah` works because both files are in the same folder.

4.12.7 A tempting wrong solution

Listing 4.49 tries the three ways of turning 3.2 into a whole number, and prints with an Indonesian locale.

Listing 4.49 `WrongReceipt.java`: truncating or rounding the weight, and formatting with an Indonesian locale.

```
import java.util.Locale;

/** Charging by truncated kilograms, and formatting with the computer's own locale. */
public class WrongReceipt {
    public static void main(String[] args) {
        double kg = 3.2;
        System.out.println("(int) kg:        " + (int) kg + " kg -> Rp" + (int) kg * 7000);
        System.out.println("Math.round(kg):  " + Math.round(kg) + " kg");
        System.out.println("Math.ceil(kg):   " + (int) Math.ceil(kg) + " kg");
        Locale indonesia = Locale.forLanguageTag("id-ID");
        System.out.println("Indonesian locale: " + String.format(indonesia, "%4.1f kg",
kg));
    }
}
```

```
(int) kg:        3 kg -> Rp21000
Math.round(kg):  3 kg
Math.ceil(kg):   4 kg
Indonesian locale:  3,2 kg
```

The cast `(int) 3.2` simply cuts off the fraction and charges 3 kg — Rp21.000 instead of Rp28.000, a loss of Rp7.000 on one bag. `Math.round` gives 3 as well; only `Math.ceil` gives the 4 the rule demands. The last line shows the locale trap: the same format prints `3,2` on a computer set to Indonesian. **Listing 4.50** adds a Python surprise.

Listing 4.50 `wrong_receipt.py`: truncation, rounding, and Python's round-half-to-even.

```
"""Charging by truncated or rounded kilograms instead of started kilograms."""
import math

kg = 3.2
print(f"int(kg):       {int(kg)} kg -> Rp{int(kg) * 7000}")
print(f"round(kg):     {round(kg)} kg")
print(f"round(2.5):    {round(2.5)} kg")
print(f"math.ceil(kg): {math.ceil(kg)} kg")
```

```
int(kg):       3 kg -> Rp21000
round(kg):     3 kg
round(2.5):    2 kg
math.ceil(kg): 4 kg
```

`round(2.5)` is 2 in Python — the round-half-to-even rule we met in Problem 4.2 — so a laundry that "rounds" 2.5 kg would charge 2 kg. The lesson: when a business rule says "per started kilogram", use the function that means exactly that, the ceiling.

> **CHECK YOURSELF.** Add a minimum charge: any order whose total is below Rp20.000 is charged Rp20.000. Where should the rule live, and what is the total for a single, non-express line of 1.5 kg of ironing?
>
> *Answer.* The rule concerns the whole receipt, not one line, so it belongs in `getTotal()`: `if (total.getAmount() < 20000) { return new Rupiah(20000); } return total;`. For 1.5 kg of ironing, 2 started kilograms cost $2 \cdot 5000 = 10000$, which is below the minimum, so the total is Rp20.000. (`toString` should then print `getTotal()` rather than the field `total`.)

4.13 Problem 4.11 — Tidying a Name

4.13.1 The problem

> **PROBLEM 4.11.** At the counter of Profitina Laundry, names are typed in a hurry: `"  siti NUR aisyah "`. Write a class `PersonName` that keeps the raw text and offers three methods: `wordCount()`, the number of words; `initials()`, the first letter of each word in capitals, each followed by a dot (`"S.N.A."`); and `tidy()`, the name with single spaces between the words, no spaces at the ends, and each word capitalized (`"Siti Nur Aisyah"`). Use only `length`, `charAt` and the character methods of Chapter 3.

4.13.2 Understanding it by hand

The whole problem rests on one definition: where does a word *start*? A character starts a word if it is not a space, and it is either the very first character or the character before it is a space. Let us mark the positions of `"  siti  NUR aisyah "` (the first character has position 0):

```
position:  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
character: _ _ s i t i _ _ _ N  U  R  _  a  i  s  y  a  h  _
starts?          ^              ^           ^
```

(Here `_` marks a space.) Word starts are at positions 2, 9 and 13, so there are three words, and the initials are `s`, `N`, `a` in capitals: `S.N.A.`. For `tidy`, every character is treated in one of three ways: a word start is capitalized and, unless it is the first word, preceded by a single space; any other letter is made small; and a space is dropped (the single space is re-created at the next word start). Edge examples: the empty name has no words and the initials `""`; a name of three spaces has no words.

4.13.3 Classes, fields and methods

```
+----------------------------------------+
| PersonName                             |
+----------------------------------------+
| - raw : String                         |
+----------------------------------------+
| + PersonName(raw : String)             |
| + wordCount() : int                    |
| + initials() : String                  |
| + tidy() : String                      |
| - startsWord(i : int) : boolean        |
+----------------------------------------+
```

The definition "starts a word" is used three times, so it becomes one private method, `startsWord(i)`: a rule written once cannot be written three slightly different ways.

4.13.4 The Java solution

Listing 4.51 is the class and **Listing 4.52** its driver.

Listing 4.51 `PersonName.java`: word starts found character by character.

```
/**
 * Problem 4.11: a person's name as typed by a customer, possibly with
 * extra spaces and random capitals. A word starts at a character that is
 * not a space and is either the first character or follows a space.
 */
public class PersonName {
    private String raw;

    public PersonName(String raw) {
        this.raw = raw;
    }

    private boolean startsWord(int i) {
        return raw.charAt(i) != ' ' && (i == 0 || raw.charAt(i - 1) == ' ');
    }

    public int wordCount() {
        int count = 0;
        for (int i = 0; i < raw.length(); i++) {
            if (startsWord(i)) {
                count++;
            }
        }
        return count;
    }

    public String initials() {
        String result = "";
        for (int i = 0; i < raw.length(); i++) {
            if (startsWord(i)) {
                result = result + Character.toUpperCase(raw.charAt(i)) + ".";
            }
        }
        return result;
    }
```

Listing 4.51 (Continued) *PersonName.java: word starts found character by character.*

```
    // "  siti   NUR aisyah " becomes "Siti Nur Aisyah".
    public String tidy() {
        String result = "";
        for (int i = 0; i < raw.length(); i++) {
            char c = raw.charAt(i);
            if (startsWord(i)) {
                if (result.length() > 0) {
                    result = result + " ";
                }
                result = result + Character.toUpperCase(c);
            } else if (c != ' ') {
                result = result + Character.toLowerCase(c);
            }
        }
        return result;
    }
}
```

Listing 4.52 `PersonNameTest.java`: the messy name and three edge cases.

```java
/** Plain test driver for PersonName: run it with "java PersonNameTest". */
public class PersonNameTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        PersonName name = new PersonName("  siti   NUR aisyah ");
        System.out.println("[" + name.tidy() + "] " + name.initials() + " " +
name.wordCount() + " words");

        check("tidy", "Siti Nur Aisyah", name.tidy());
        check("initials", "S.N.A.", name.initials());
        check("word count", 3, name.wordCount());
        check("one word", "B.", new PersonName("budi").initials());
        check("empty name", "", new PersonName("").initials());
        check("only spaces", 0, new PersonName("   ").wordCount());
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

```
[Siti Nur Aisyah] S.N.A. 3 words
PASS tidy
PASS initials
PASS word count
PASS one word
PASS empty name
PASS only spaces
6 passed, 0 failed
```

The square brackets in the first line are there to make leading and trailing spaces visible: there are none.

4.13.5 The Python solution

Listing 4.53 uses indexing, `self._raw[i]`, where Java uses `charAt(i)`.

Listing 4.53 `person_name.py`: Problem 4.11 in Python.

```
"""Problem 4.11: a person's name as typed by a customer, possibly with
extra spaces and random capitals. A word starts at a character that is
not a space and is either the first character or follows a space.
"""
import unittest


class PersonName:
    def __init__(self, raw):
        self._raw = raw

    def _starts_word(self, i):
        return self._raw[i] != " " and (i == 0 or self._raw[i - 1] == " ")

    def word_count(self):
        count = 0
        for i in range(len(self._raw)):
            if self._starts_word(i):
                count += 1
        return count

    def initials(self):
        result = ""
        for i in range(len(self._raw)):
            if self._starts_word(i):
                result = result + self._raw[i].upper() + "."
        return result

    def tidy(self):
        # "  siti   NUR aisyah " becomes "Siti Nur Aisyah".
        result = ""
        for i in range(len(self._raw)):
            c = self._raw[i]
            if self._starts_word(i):
                if len(result) > 0:
                    result = result + " "
                result = result + c.upper()
            elif c != " ":
                result = result + c.lower()
        return result
```

Listing 4.53 (Continued) *person_name.py: Problem 4.11 in Python.*

```
class TestPersonName(unittest.TestCase):
    def test_messy_name(self):
        name = PersonName("  siti   NUR aisyah ")
        self.assertEqual(name.tidy(), "Siti Nur Aisyah")
        self.assertEqual(name.initials(), "S.N.A.")
        self.assertEqual(name.word_count(), 3)

    def test_small_cases(self):
        self.assertEqual(PersonName("budi").initials(), "B.")
        self.assertEqual(PersonName("").initials(), "")
        self.assertEqual(PersonName("   ").word_count(), 0)


if __name__ == "__main__":
    name = PersonName("  siti   NUR aisyah ")
    print(f"[{name.tidy()}] {name.initials()} {name.word_count()} words")
    unittest.main(verbosity=2)
```

```
[Siti Nur Aisyah] S.N.A. 3 words
test_messy_name (__main__.TestPersonName.test_messy_name) ... ok
test_small_cases (__main__.TestPersonName.test_small_cases) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

4.13.6 Line by line

- `return raw.charAt(i) != ' ' && (i == 0 || raw.charAt(i - 1) == ' ');` — the definition, word for word. The order inside the parentheses matters: when `i` is 0, `i == 0` is already `true`, and Java does not evaluate the right side of `||`, so `raw.charAt(-1)`, which would crash, is never attempted. This *short-circuit* evaluation is a feature you can rely on in both Java and Python.
- `' '` in single quotes is a `char`, one character; `" "` in double quotes is a `String`. `charAt` returns a `char`, so it is compared with `' '`. Python has no separate character type: `self._raw[i]` is a string of length one, compared with `" "`.
- `Character.toUpperCase(c)` and `Character.toLowerCase(c)` in Java, `c.upper()` and `c.lower()` in Python, change one letter and leave other characters alone.
- `if (result.length() > 0) { result = result + " "; }` — a space goes *before* every word except the first, which is simpler than trying to remove a trailing space at the end.

4.13.7 A tempting wrong solution

Both languages have a method that splits a string into words, and it is tempting to use it with a single space as the separator. **Listing 4.54** shows what happens. (The loop `for (String word : …)` visits each piece in turn; such arrays of pieces are the subject of Chapter 5.)

Listing 4.54 `WrongInitials.java`: `split(" ")` produces empty words, and `charAt(0)` of an empty word crashes.

```java
/** split(" ") turns every extra space into an empty word, and "".charAt(0) fails. */
public class WrongInitials {
    public static void main(String[] args) {
        String raw = "  siti   NUR aisyah ";
        String initials = "";
        for (String word : raw.split(" ")) {
            System.out.print("[" + word + "] ");
        }
        System.out.println();
        for (String word : raw.split(" ")) {
            initials = initials + Character.toUpperCase(word.charAt(0)) + ".";
        }
        System.out.println(initials);
    }
}
```

```
[] [] [siti] [] [] [NUR] [aisyah]
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Index 0 out of bounds
for length 0
    at java.base/jdk.internal.util.Preconditions$1.apply(Preconditions.java:55)
    at java.base/jdk.internal.util.Preconditions$1.apply(Preconditions.java:52)
    at java.base/jdk.internal.util.Preconditions$4.apply(Preconditions.java:213)
    at java.base/jdk.internal.util.Preconditions$4.apply(Preconditions.java:210)
    at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:98)
    at
java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:106)
    at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:302)
    at java.base/java.lang.String.checkIndex(String.java:4822)
    at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:46)
    at java.base/java.lang.String.charAt(String.java:1544)
    at WrongInitials.main(WrongInitials.java:11)
```

Splitting at *every* single space produces an empty piece between two neighboring spaces and before a leading space. The first empty piece has no character at position 0, and the program stops with a `StringIndexOutOfBoundsException`. The last line of the long message, `at WrongInitials.main(WrongInitials.java:11)`, names the file and the line: read the bottom line of your own code in such a *stack trace* first. Python fails in the same way (**Listing 4.55**).

Listing 4.55 `wrong_initials.py`: the same empty words in Python, and an `IndexError`.

```
"""split(" ") turns every extra space into an empty word, and ""[0] fails."""
raw = "  siti   NUR aisyah "
print(raw.split(" "))
initials = ""
for word in raw.split(" "):
    initials = initials + word[0].upper() + "."
print(initials)
```

```
['', '', 'siti', '', '', 'NUR', 'aisyah', '']
Traceback (most recent call last):
  File "/home/reader/Chapter04/python/wrong_initials.py", line 6, in <module>
    initials = initials + word[0].upper() + "."
                          ~~~~^^^
IndexError: string index out of range
```

(The path in the `File` line is wherever the file lives on your computer.) The printed list shows the culprits: `''` is the empty string. Both languages *do* have a correct shortcut — Python's `split()` without an argument, and Java's `trim().split("\\s+")` — but you should understand the character-by-character version first, because it is what those shortcuts do inside.

> **CHECK YOURSELF.** Add a method `lastNameFirst()` that returns the tidy name with the last word first, followed by a comma: `"Aisyah, Siti Nur"`. What does it return for `"budi"`, a single word?
>
> *Answer.* Take `String t = tidy();`, find the last space with `int k = t.lastIndexOf(' ');` and, if `k` is −1 (no space), return `t` unchanged; otherwise return `t.substring(k + 1) + ", " + t.substring(0, k)`. For `"budi"` there is no space, so the result is `"Budi"`. In Python, `t.rfind(" ")` plays the role of `lastIndexOf`, and slices `t[k + 1:]` and `t[:k]` the role of `substring`.

4.14 Problem 4.12 — Password Rules

4.14.1 The problem

> **PROBLEM 4.12.** Before `changePassword` of Problem 4.3 accepts a new password, it should check it. Write a class `PasswordChecker` with a static method `problems(password)` that returns a text listing every rule the password breaks, separated by `"; "`, or the empty string if it breaks none, and a static method `isStrong(password)` that returns `true` exactly when there are no problems. The rules: at least 8 characters (`"too short"`), at least one digit (`"no digit"`), at least one capital letter (`"no capital letter"`), at least one small letter (`"no small letter"`).

4.14.2 Understanding it by hand

`sesame`: 6 characters, too short; no digit; no capital; it has small letters. So: `too short; no digit; no capital letter`. `1sesame`: 7 characters, still too short; the digit `1` counts; no capital. So: `too short; no capital letter`. `Sesame2026`: 10 characters, a capital `S`, small letters, digits — strong. The edge cases sit at the length boundary: `Abcdefg1` has exactly 8 characters and is strong; `Abcdef1` has 7 and is only too short. And the empty password breaks all four rules.

Notice *how* we checked `1sesame` for a digit: we looked at *every* character, and the verdict "has a digit" was already settled at the first one, while the verdict "has no capital" could only be given after the *last* one. A loop that decides "some character has property P" may stop as soon as it finds one; a loop that decides "no character has P" must look at them all.

4.14.3 Classes, fields and methods

```
+-----------------------------------------------------+
| PasswordChecker                                     |
+-----------------------------------------------------+
| + MIN_LENGTH : int = 8                              |
+-----------------------------------------------------+
| + problems(password : String) : String   (static)   |
| + isStrong(password : String) : boolean  (static)   |
| - add(report, problem) : String          (static)   |
+-----------------------------------------------------+
```

The plan for `problems`: one loop that *counts* digits, capitals and small letters; then, *after* the loop, four `if` statements that turn the counts into the report.

4.14.4 The Java solution

Listing 4.56 is the checker and **Listing 4.57** its driver.

Listing 4.56 `PasswordChecker.java`: count in one loop, decide after it.

```
/**
 * Problem 4.12: checks a proposed password against four rules and
 * explains every rule it breaks. One pass over the characters counts
 * digits, capitals and small letters; the verdict comes after the loop.
 */
public class PasswordChecker {
    public static final int MIN_LENGTH = 8;

    // Appends one more problem to the report, with "; " between problems.
    private static String add(String report, String problem) {
        if (report.isEmpty()) {
            return problem;
        }
        return report + "; " + problem;
    }

    public static String problems(String password) {
        int digits = 0;
        int upper = 0;
        int lower = 0;
        for (int i = 0; i < password.length(); i++) {
            char c = password.charAt(i);
            if (Character.isDigit(c)) {
                digits++;
            } else if (Character.isUpperCase(c)) {
                upper++;
            } else if (Character.isLowerCase(c)) {
                lower++;
            }
        }
        String report = "";
        if (password.length() < MIN_LENGTH) {
            report = add(report, "too short");
        }
        if (digits == 0) {
            report = add(report, "no digit");
        }
        if (upper == 0) {
            report = add(report, "no capital letter");
        }
        if (lower == 0) {
            report = add(report, "no small letter");
        }
        return report;
    }
```

Listing 4.56 (Continued) *PasswordChecker.java: count in one loop, decide after it.*

```
    public static boolean isStrong(String password) {
        return problems(password).isEmpty();
    }
}
```

Listing 4.57 `PasswordCheckerTest.java`: digits at either end, the length boundary, and the empty password.

```java
/** Plain test driver for PasswordChecker: run it with "java PasswordCheckerTest". */
public class PasswordCheckerTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        System.out.println("sesame     -> " + PasswordChecker.problems("sesame"));
        System.out.println("1sesame    -> " + PasswordChecker.problems("1sesame"));
        System.out.println("Sesame2026 -> " + PasswordChecker.isStrong("Sesame2026"));

        check("strong", true, PasswordChecker.isStrong("Sesame2026"));
        check("digit only at the start still counts", "too short; no capital letter",
                PasswordChecker.problems("1sesame"));
        check("digit only at the end still counts", true,
PasswordChecker.isStrong("Laundry7"));
        check("empty password", "too short; no digit; no capital letter; no small letter",
                PasswordChecker.problems(""));
        check("exactly eight is long enough", true, PasswordChecker.isStrong("Abcdefg1"));
        check("seven is too short", "too short", PasswordChecker.problems("Abcdef1"));
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

```
sesame     -> too short; no digit; no capital letter
1sesame    -> too short; no capital letter
Sesame2026 -> true
PASS strong
PASS digit only at the start still counts
PASS digit only at the end still counts
PASS empty password
PASS exactly eight is long enough
PASS seven is too short
6 passed, 0 failed
```

4.14.5 The Python solution

In **Listing 4.58** the two functions live at the top level of the module, the Python way of writing what Java calls static methods.

Listing 4.58 `password_checker.py`: Problem 4.12 in Python.

```
"""Problem 4.12: checks a proposed password against four rules and
explains every rule it breaks. One pass over the characters counts
digits, capitals and small letters; the verdict comes after the loop.
"""
import unittest

MIN_LENGTH = 8


def _add(report, problem):
    # Appends one more problem to the report, with "; " between problems.
    if report == "":
        return problem
    return report + "; " + problem


def problems(password):
    digits = 0
    upper = 0
    lower = 0
    for c in password:
        if c.isdigit():
            digits += 1
        elif c.isupper():
            upper += 1
        elif c.islower():
            lower += 1
    report = ""
    if len(password) < MIN_LENGTH:
        report = _add(report, "too short")
    if digits == 0:
        report = _add(report, "no digit")
    if upper == 0:
        report = _add(report, "no capital letter")
    if lower == 0:
        report = _add(report, "no small letter")
    return report


def is_strong(password):
    return problems(password) == ""
```

Listing 4.58 (Continued) *password_checker.py: Problem 4.12 in Python.*

```
class TestPasswordChecker(unittest.TestCase):
    def test_strong(self):
        self.assertTrue(is_strong("Sesame2026"))
        self.assertTrue(is_strong("Laundry7"))
        self.assertTrue(is_strong("Abcdefg1"))

    def test_reports(self):
        self.assertEqual(problems("1sesame"), "too short; no capital letter")
        self.assertEqual(problems(""), "too short; no digit; no capital letter; no small
letter")
        self.assertEqual(problems("Abcdef1"), "too short")


if __name__ == "__main__":
    print(f"sesame     -> {problems('sesame')}")
    print(f"1sesame    -> {problems('1sesame')}")
    print(f"Sesame2026 -> {is_strong('Sesame2026')}")
    unittest.main(verbosity=2)
```

```
sesame     -> too short; no digit; no capital letter
1sesame    -> too short; no capital letter
Sesame2026 -> True
test_reports (__main__.TestPasswordChecker.test_reports) ... ok
test_strong (__main__.TestPasswordChecker.test_strong) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Java prints the Boolean value as `true` and Python as `True`; the value is the same.

4.14.6 Line by line

- `if (Character.isDigit(c)) { digits++; } else if (Character.isUpperCase(c)) { upper++; } else if (…)` — a character is at most one of the three kinds, so an `else if` chain is right, and each counter grows with `++`, "add one". Characters that are none of the three, such as `!`, are simply not counted.
- `private static String add(String report, String problem)` — the separator `"; "` is needed *between* problems, not before the first; the helper decides that in one place, so the four `if` statements stay simple.
- `return problems(password).isEmpty();` — `isStrong` is defined through `problems`, so the two can never disagree. (Compare Problem 4.5: store, or here compute, each fact once.)
- Python's `for c in password:` visits the characters directly, without an index; `c.isdigit()`, `c.isupper()` and `c.islower()` are the character tests.

A word of honesty about the rules. They are good for practicing loops, but modern guidance on real passwords, such as NIST Special Publication 800-63B, advises *against* forcing mixtures of character kinds and recommends long passwords checked against lists of commonly used ones instead [9]. A real system would also never store or print the password itself, only a salted hash of it.

4.14.7 A tempting wrong solution

Listing 4.59 `WrongPassword.java`: a loop that returns a verdict after the first character.

```java
/** A loop that decides after looking at the first character only. */
public class WrongPassword {
    public static boolean hasDigit(String s) {
        for (int i = 0; i < s.length(); i++) {
            if (Character.isDigit(s.charAt(i))) {
                return true;
            } else {
                return false; // decides too early
            }
        }
        return false;
    }

    public static void main(String[] args) {
        System.out.println("1sesame:  " + hasDigit("1sesame"));
        System.out.println("Laundry7: " + hasDigit("Laundry7"));
    }
}
```

```
1sesame:  true
Laundry7: false
```

The `else { return false; }` inside the loop ends the method after looking at the *first* character. `1sesame` passes because its digit happens to come first; `Laundry7` fails, although it has a digit. The loop body runs at most once, so it is not really a loop at all. The debugger makes this visible immediately: step through `hasDigit("Laundry7")` and watch `i` never reach 1. **Listing 4.60** shows the same bug in Python.

Listing 4.60 `wrong_password.py`: the same early return in Python.

```
"""A loop that decides after looking at the first character only."""


def has_digit(s):
    for c in s:
        if c.isdigit():
            return True
        else:
            return False  # decides too early
    return False


print(f"1sesame:  {has_digit('1sesame')}")
print(f"Laundry7: {has_digit('Laundry7')}")
```

```
1sesame:  True
Laundry7: False
```

The fix is to move `return false` *after* the loop: "if no character was a digit, then there is no digit".

> **CHECK YOURSELF.** Add a fifth rule, `"no symbol"`: the password must contain at least one character that is neither a letter nor a digit. Which of `Sesame2026` and `Sesame-2026` are strong under five rules?
>
> *Answer.* Add a counter `symbols` and a final `else { symbols++; }` to the chain inside the loop (in Java, `Character.isLetterOrDigit(c)` could be used instead), and after the loop `if (symbols == 0) { report = add(report, "no symbol"); }`. Then `Sesame2026` reports `no symbol`, and `Sesame-2026` is strong.

4.15 Problem 4.13 — Tracing Loops by Hand

4.15.1 The problem

> **PROBLEM 4.13.** Without running a program, trace the following two loops with a table that has one row per iteration: (a) the loop that computes the sum of the digits of $n = 4071$ by repeatedly taking the last digit `n % 10` and dropping it with `n / 10`; (b) Euclid's algorithm for the greatest common divisor of $a = 84$ and $b = 36$, which repeats "$r = a \bmod b$; $a = b$; $b = r$" while $b \neq 0$ and then returns a. Then write a program whose trace output reproduces your tables, and test both methods.

4.15.2 Understanding it by hand

A *trace table* has a column for every variable that changes and a row for every pass through the loop, filled in *after* the body has run once. It is the paper version of the debugger's watch window, and it is the single best way to understand a loop.

(a) Start with `total = 0` and `n = 4071`. In each pass, `digit = n % 10` is the last digit, `total` grows by it, and `n / 10` (integer division) drops it.

Table 4.6 Trace of the digit-sum loop for $n = 4071$: four passes, one per digit, ending when n reaches 0.

Step	`digit` = n mod 10	`total`	n After `n / 10`
1	$4071 \bmod 10 = 1$	$0 + 1 = 1$	407
2	$407 \bmod 10 = 7$	$1 + 7 = 8$	40
3	$40 \bmod 10 = 0$	$8 + 0 = 8$	4
4	$4 \bmod 10 = 4$	$8 + 4 = 12$	0

After step 4 the condition `n > 0` is false and the loop stops: the digit sum is 12, and indeed $4 + 0 + 7 + 1 = 12$. The loop runs once per digit, so a d-digit number takes d passes. For $n = 0$ it runs zero times and returns 0.

(b) Euclid's algorithm rests on one fact: any number that divides both a and b also divides a mod b (which is a minus a multiple of b), and conversely, so

$$\gcd(a, b) = \gcd(b,\ a \bmod b), \qquad \gcd(a, 0) = a \tag{4.8}$$

Each pass of the loop applies Equation (4.8) once [10]:

Table 4.7 Trace of Euclid's algorithm for $a = 84$, $b = 36$: after two passes $b = 0$ and the answer is $a = 12$.

a	b	r=a mod b	New a, new b
84	36	$84 - 2 \cdot 36 = 12$	36, 12
36	12	$36 - 3 \cdot 12 = 0$	12, 0

So gcd(84,36) = 12; check: $84 = 7 \cdot 12$ and $36 = 3 \cdot 12$, and 7 and 3 have no common factor. If we start with the smaller number first, gcd(36,84), the first pass gives $r = 36 \bmod 84 = 36$ and simply swaps the two numbers — one wasted pass, but the same answer.

4.15.3 Classes, fields and methods

There is nothing to remember between calls, so `LoopTrace` holds two static methods, each with a Boolean parameter `trace` that switches the table output on or off. The same method is thus both *explained* (with `trace = true`) and *tested* (with `trace = false`).

```
+-------------------------------------------------+
| LoopTrace                                       |
+-------------------------------------------------+
| + digitSum(n : int, trace : boolean) : int      |
| + gcd(a : int, b : int, trace : boolean) : int  |
| + main(args)                                    |
+-------------------------------------------------+
```

4.15.4 The Java solution

Running **Listing 4.61** prints both tables.

Listing 4.61 `LoopTrace.java`: two loops that print their own trace tables.

```java
/**
 * Problem 4.13: two while loops that are traced by hand in the text.
 * With trace = true each loop prints one row per iteration, so the
 * printed table can be compared with the table written on paper.
 */
public class LoopTrace {

    public static int digitSum(int n, boolean trace) {
        int total = 0;
        int step = 0;
        if (trace) {
            System.out.println("step  digit  total     n");
        }
        while (n > 0) {
            int digit = n % 10; // the last digit
            total = total + digit;
            n = n / 10;         // drop the last digit
            step++;
            if (trace) {
                System.out.printf("%4d %6d %6d %5d%n", step, digit, total, n);
            }
        }
        return total;
    }

    // Euclid's algorithm: gcd(a, b) = gcd(b, a % b), and gcd(a, 0) = a.
    public static int gcd(int a, int b, boolean trace) {
        if (trace) {
            System.out.println("   a    b  a % b");
        }
        while (b != 0) {
            int r = a % b;
            if (trace) {
                System.out.printf("%4d %4d %6d%n", a, b, r);
            }
            a = b;
            b = r;
        }
        return a;
    }

    public static void main(String[] args) {
        System.out.println("digitSum(4071) = " + digitSum(4071, true));
        System.out.println("gcd(84, 36) = " + gcd(84, 36, true));
    }
}
```

```
step  digit  total     n
   1      1      1   407
   2      7      8    40
   3      0      8     4
   4      4     12     0
digitSum(4071) = 12
   a    b  a % b
  84   36     12
  36   12      0
gcd(84, 36) = 12
```

The printed rows are **Table 4.6** and **Table 4.7**. Notice that each table is printed *before* the line `digitSum(4071) = 12`: to build that line, Java must first call `digitSum`, and the call prints its rows on the way. **Listing 4.62** checks the edge cases.

Listing 4.62 `LoopTraceTest.java`: zero, one digit, trailing zeros, and both orders for Euclid.

```java
/** Plain test driver for LoopTrace: run it with "java LoopTraceTest". */
public class LoopTraceTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        check("digitSum(4071)", 12, LoopTrace.digitSum(4071, false));
        check("digitSum(9)", 9, LoopTrace.digitSum(9, false));
        check("digitSum(0) runs zero times", 0, LoopTrace.digitSum(0, false));
        check("digitSum(1000)", 1, LoopTrace.digitSum(1000, false));
        check("gcd(84, 36)", 12, LoopTrace.gcd(84, 36, false));
        check("gcd(36, 84) swaps first", 12, LoopTrace.gcd(36, 84, false));
        check("gcd(17, 5)", 1, LoopTrace.gcd(17, 5, false));
        check("gcd(5, 0)", 5, LoopTrace.gcd(5, 0, false));
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

```
PASS digitSum(4071)
PASS digitSum(9)
PASS digitSum(0) runs zero times
PASS digitSum(1000)
PASS gcd(84, 36)
PASS gcd(36, 84) swaps first
PASS gcd(17, 5)
PASS gcd(5, 0)
8 passed, 0 failed
```

4.15.5 The Python solution

Listing 4.63 prints the same two tables.

Listing 4.63 `loop_trace.py`: Problem 4.13 in Python, with `//` for integer division.

```
"""Problem 4.13: two while loops that are traced by hand in the text.
With trace=True each loop prints one row per iteration, so the
printed table can be compared with the table written on paper.
"""
import unittest


def digit_sum(n, trace):
    total = 0
    step = 0
    if trace:
        print("step  digit  total     n")
    while n > 0:
        digit = n % 10  # the last digit
        total = total + digit
        n = n // 10     # drop the last digit (// keeps n a whole number)
        step += 1
        if trace:
            print(f"{step:4d} {digit:6d} {total:6d} {n:5d}")
    return total


def gcd(a, b, trace):
    # Euclid's algorithm: gcd(a, b) = gcd(b, a % b), and gcd(a, 0) = a.
    if trace:
        print("   a    b  a % b")
    while b != 0:
        r = a % b
        if trace:
            print(f"{a:4d} {b:4d} {r:6d}")
        a = b
        b = r
    return a


class TestLoopTrace(unittest.TestCase):
    def test_digit_sum(self):
        self.assertEqual(digit_sum(4071, False), 12)
        self.assertEqual(digit_sum(9, False), 9)
        self.assertEqual(digit_sum(0, False), 0)
        self.assertEqual(digit_sum(1000, False), 1)
```

Listing 4.63 (Continued) *loop_trace.py: Problem 4.13 in Python, with // for integer division.*

```
    def test_gcd(self):
        self.assertEqual(gcd(84, 36, False), 12)
        self.assertEqual(gcd(36, 84, False), 12)
        self.assertEqual(gcd(17, 5, False), 1)
        self.assertEqual(gcd(5, 0, False), 5)


if __name__ == "__main__":
    print(f"digit_sum(4071) = {digit_sum(4071, True)}")
    print(f"gcd(84, 36) = {gcd(84, 36, True)}")
    unittest.main(verbosity=2)
```

```
step  digit  total     n
   1      1      1   407
   2      7      8    40
   3      0      8     4
   4      4     12     0
digit_sum(4071) = 12
   a    b  a % b
  84   36     12
  36   12      0
gcd(84, 36) = 12
test_digit_sum (__main__.TestLoopTrace.test_digit_sum) ... ok
test_gcd (__main__.TestLoopTrace.test_gcd) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

4.15.6 Line by line

- `while (n > 0)` — the condition is checked *before* each pass, so for `n = 0` the body never runs; that is why `digitSum(0)` is correctly 0.
- `n = n / 10;` — integer division in Java because both operands are `int`. In Python the same step *must* be written `n // 10`; see below.
- `System.out.printf("%4d %6d %6d %5d%n", …)` — `printf` is `String.format` and `print` in one; the widths line the numbers up under the header.
- In `gcd`, the three assignments `r = a % b; a = b; b = r;` must happen in this order: the helper variable `r` keeps the remainder safe while `a` and `b` move along.
- We call the running sum `total`, not `sum`, in both languages: in Python `sum` is the name of a built-in function, and using it as a variable hides that function.

4.15.7 A tempting wrong solution

Listing 4.64 changes a single character of the condition.

Listing 4.64 `WrongLoop.java`: the condition `n > 9` stops one pass too early.

```java
/** An off-by-one loop condition: the last remaining digit is never added. */
public class WrongLoop {
    public static void main(String[] args) {
        int n = 4071;
        int total = 0;
        while (n > 9) {
            total = total + n % 10;
            n = n / 10;
        }
        System.out.println("digit sum of 4071: " + total);
    }
}
```

```
digit sum of 4071: 8
```

The condition `n > 9` feels right — "while there is more than one digit" — but it stops when $n = 4$, before the last digit is added: the answer is 8 instead of 12. Trace tables catch off-by-one errors like this at once: the table has three rows instead of four. **Listing 4.65** repeats it in Python and adds a second, Python-specific trap.

Listing 4.65 `wrong_loop.py`: the off-by-one condition, and `/` where `//` was meant.

```
"""Two broken digit sums: an off-by-one condition, and / where // was meant."""
n = 4071
total = 0
while n > 9:
    total = total + n % 10
    n = n // 10
print(f"n > 9 instead of n > 0: {total}")

n = 4071
total = 0
steps = 0
while n > 0:
    total = total + n % 10
    n = n / 10  # true division: n never becomes exactly 0 quickly
    steps += 1
print(f"/ instead of //: total {total} after {steps} steps")
```

```
n > 9 instead of n > 0: 8
/ instead of //: total 13.333333333333357 after 328 steps
```

With `/`, Python divides exactly: $4071/10 = 407.1$, then 40.71, and so on. The number never becomes exactly 0 in a few steps; it only shrinks until it is too small for a `double` to represent, which takes 328 passes, while the "digits" `n % 10` are fractions that add up to nonsense. This is the mirror image of Java's integer-division trap of Problem 4.1: in Java, `/` between integers is sometimes a surprise; in Python, `/` between integers is *never* integer division.

> **CHECK YOURSELF.** Trace `gcd(1071, 462)` by hand. How many passes does the loop make, and what is the result?
>
> *Answer.* $1071 \bmod 462 = 1071 - 2 \cdot 462 = 147$, so (a, b) becomes (462,147); $462 \bmod 147 = 462 - 3 \cdot 147 = 21$, so (147,21); $147 \bmod 21 = 0$, so (21,0). Three passes, and $\gcd(1071,462) = 21$. Check: $1071 = 51 \cdot 21$ and $462 = 22 \cdot 21$, and 51 and 22 have no common factor.

4.16 Problem 4.14 — Leap Years and the Next Day

4.16.1 The problem

> **PROBLEM 4.14.** Write a class `SimpleDate` for a date in the Gregorian calendar with fields year, month and day. It has a static method `isLeapYear(year)`, a static method `daysInMonth(year, month)`, a method `isValid()` that says whether the date exists, a method `nextDay()` that returns a new `SimpleDate` for the following day, and a `toString` in the international format `YYYY-MM-DD`, such as `2024-02-29`. Also count, with a loop, the leap years from 1901 to 2100.

4.16.2 Understanding it by hand

The Gregorian rule has three parts [11]: a year divisible by 4 is a leap year, *except* a year divisible by 100, which is not, *except* a year divisible by 400, which is. As one Boolean expression:

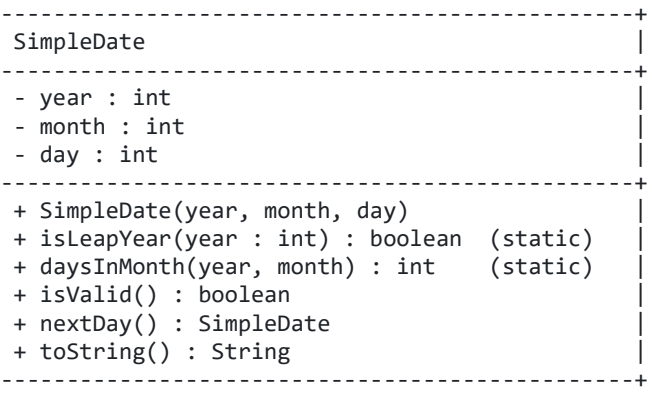
$$\text{leap}(y) = (4 \mid y \ \wedge \ 100 \nmid y) \ \vee \ 400 \mid y \tag{4.9}$$

where $4 \mid y$ reads "4 divides y", that is, $y \bmod 4 = 0$; $\nmid$ means "does not divide"; $\wedge$ is "and" and $\vee$ is "or". So 2024 is a leap year ($4 \mid 2024$, $100 \nmid 2024$), 2023 is not, 1900 is not ($100 \mid 1900$ but $400 \nmid 1900$), and 2000 is ($400 \mid 2000$).

Counting from 1901 to 2100: the multiples of 4 are 1904,1908, … ,2100, which is $\frac{2100-1904}{4} + 1 = 49 + 1 = 50$ years. Of these, the century years 2000 and 2100 need a second look: 2000 is divisible by 400 and stays a leap year, but 2100 is not. So the answer is $50 - 1 = 49$.

For `nextDay`, there are three cases, checked in order: if the day is not the last of its month, add one to the day; otherwise, if the month is not December, go to day 1 of the next month; otherwise go to 1 January of the next year. So 28 February 2024 is followed by 29 February 2024, 28 February 2023 by 1 March 2023, and 31 December 2026 by 1 January 2027.

4.16.3 Classes, fields and methods



`isLeapYear` and `daysInMonth` are static because they are facts about the calendar, not about one particular date.

4.16.4 The Java solution

Listing 4.66 is the class; **Listing 4.67** first counts the leap years with a `for` loop and then checks the hand examples.

Listing 4.66 `SimpleDate.java`: leap years by Equation (4.9), month lengths by `switch`, and the next day in three cases.

```
/**
 * Problem 4.14: a calendar date in the Gregorian calendar. A year is a
 * leap year if it is divisible by 4, except century years, which must
 * be divisible by 400: 2024 and 2000 are leap years, 1900 and 2100 are not.
 */
public class SimpleDate {
    private int year;
    private int month;
    private int day;

    public SimpleDate(int year, int month, int day) {
        this.year = year;
        this.month = month;
        this.day = day;
    }

    public static boolean isLeapYear(int year) {
        return (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;
    }

    public static int daysInMonth(int year, int month) {
        switch (month) {
            case 2:
                return isLeapYear(year) ? 29 : 28;
            case 4:
            case 6:
            case 9:
            case 11:
                return 30;
            default:
                return 31;
        }
    }

    public boolean isValid() {
        return month >= 1 && month <= 12 && day >= 1 && day <= daysInMonth(year, month);
    }
```

Listing 4.66 (Continued) *SimpleDate.java: leap years by Equation (4.9), month lengths by switch, and the next day in three cases.*

```
    public SimpleDate nextDay() {
        if (day < daysInMonth(year, month)) {
            return new SimpleDate(year, month, day + 1);
        }
        if (month < 12) {
            return new SimpleDate(year, month + 1, 1);
        }
        return new SimpleDate(year + 1, 1, 1);
    }

    @Override
    public String toString() {
        return String.format("%04d-%02d-%02d", year, month, day);
    }
}
```

Listing 4.67 `SimpleDateTest.java`: the four kinds of year, month ends, the year end, and invalid dates.

```java
/** Plain test driver for SimpleDate: run it with "java SimpleDateTest". */
public class SimpleDateTest {
    private static int passed = 0;
    private static int failed = 0;

    private static void check(String label, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS " + label);
        } else {
            failed++;
            System.out.println("FAIL " + label + ": expected " + expected + ", got " +
actual);
        }
    }

    public static void main(String[] args) {
        int leapYears = 0;
        for (int year = 1901; year <= 2100; year++) {
            if (SimpleDate.isLeapYear(year)) {
                leapYears++;
            }
        }
        System.out.println("Leap years from 1901 to 2100: " + leapYears);

        check("2024 is leap", true, SimpleDate.isLeapYear(2024));
        check("2023 is not", false, SimpleDate.isLeapYear(2023));
        check("1900 is not", false, SimpleDate.isLeapYear(1900));
        check("2000 is", true, SimpleDate.isLeapYear(2000));
        check("April has 30 days", 30, SimpleDate.daysInMonth(2026, 4));
        check("into 29 February", "2024-02-29", new SimpleDate(2024, 2,
28).nextDay().toString());
        check("past 28 February", "2023-03-01", new SimpleDate(2023, 2,
28).nextDay().toString());
        check("new year", "2027-01-01", new SimpleDate(2026, 12, 31).nextDay().toString());
        check("29 Feb 2023 invalid", false, new SimpleDate(2023, 2, 29).isValid());
        check("month 13 invalid", false, new SimpleDate(2026, 13, 1).isValid());
        System.out.println(passed + " passed, " + failed + " failed");
    }
}
```

```
Leap years from 1901 to 2100: 49
PASS 2024 is leap
PASS 2023 is not
PASS 1900 is not
PASS 2000 is
PASS April has 30 days
PASS into 29 February
PASS past 28 February
PASS new year
PASS 29 Feb 2023 invalid
PASS month 13 invalid
10 passed, 0 failed
```

4.16.5 The Python solution

Listing 4.68 marks `is_leap_year` and `days_in_month` with `@staticmethod`, Python's way of saying they belong to the class.

Listing 4.68 `simple_date.py`: Problem 4.14 in Python, with `match` in place of `switch`.

```
"""Problem 4.14: a calendar date in the Gregorian calendar. A year is a
leap year if it is divisible by 4, except century years, which must
be divisible by 400: 2024 and 2000 are leap years, 1900 and 2100 are not.
"""
import unittest


class SimpleDate:
    def __init__(self, year, month, day):
        self._year = year
        self._month = month
        self._day = day

    @staticmethod
    def is_leap_year(year):
        return (year % 4 == 0 and year % 100 != 0) or year % 400 == 0

    @staticmethod
    def days_in_month(year, month):
        match month:
            case 2:
                return 29 if SimpleDate.is_leap_year(year) else 28
            case 4 | 6 | 9 | 11:
                return 30
            case _:
                return 31

    def is_valid(self):
        return (1 <= self._month <= 12
                and 1 <= self._day <= SimpleDate.days_in_month(self._year, self._month))

    def next_day(self):
        if self._day < SimpleDate.days_in_month(self._year, self._month):
            return SimpleDate(self._year, self._month, self._day + 1)
        if self._month < 12:
            return SimpleDate(self._year, self._month + 1, 1)
        return SimpleDate(self._year + 1, 1, 1)

    def __str__(self):
        return f"{self._year:04d}-{self._month:02d}-{self._day:02d}"
```

Listing 4.68 (Continued) *simple_date.py: Problem 4.14 in Python, with match in place of switch.*

```
class TestSimpleDate(unittest.TestCase):
    def test_leap_years(self):
        self.assertTrue(SimpleDate.is_leap_year(2024))
        self.assertFalse(SimpleDate.is_leap_year(2023))
        self.assertFalse(SimpleDate.is_leap_year(1900))
        self.assertTrue(SimpleDate.is_leap_year(2000))

    def test_next_day(self):
        self.assertEqual(str(SimpleDate(2024, 2, 28).next_day()), "2024-02-29")
        self.assertEqual(str(SimpleDate(2023, 2, 28).next_day()), "2023-03-01")
        self.assertEqual(str(SimpleDate(2026, 12, 31).next_day()), "2027-01-01")

    def test_validity(self):
        self.assertEqual(SimpleDate.days_in_month(2026, 4), 30)
        self.assertFalse(SimpleDate(2023, 2, 29).is_valid())
        self.assertFalse(SimpleDate(2026, 13, 1).is_valid())


if __name__ == "__main__":
    leap_years = 0
    for year in range(1901, 2101):
        if SimpleDate.is_leap_year(year):
            leap_years += 1
    print(f"Leap years from 1901 to 2100: {leap_years}")
    unittest.main(verbosity=2)
```

```
Leap years from 1901 to 2100: 49
test_leap_years (__main__.TestSimpleDate.test_leap_years) ... ok
test_next_day (__main__.TestSimpleDate.test_next_day) ... ok
test_validity (__main__.TestSimpleDate.test_validity) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

4.16.6 Line by line

- `return (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;` — Equation (4.9) in Java. In Java, `&&` binds more tightly than `||`, so the parentheses are not strictly needed, but they show the reader the structure of the rule.
- `case 4: case 6: case 9: case 11: return 30;` — several `case` labels may share one body. Python writes `case 4 | 6 | 9 | 11:`.
- `return isLeapYear(year) ? 29 : 28;` — the conditional operator again; Python's form is `29 if … else 28`.
- `month >= 1 && month <= 12 && day >= 1 && day <= daysInMonth(year, month)` — the order matters: `daysInMonth` is only consulted once the month is known to be between 1 and 12. Python can write `1 <= self._month <= 12`, a *chained comparison* that Java does not have.
- `String.format("%04d-%02d-%02d", …)` — at least four digits for the year and two for month and day, padded with zeros. Dates written this way sort correctly as plain text, which is one reason the international standard uses them.

4.16.7 A tempting wrong solution

Listing 4.69 keeps only the first part of the rule.

Listing 4.69 `WrongLeapYear.java`: "divisible by 4" alone.

```java
/** "Divisible by 4" alone is the Julian rule, not the Gregorian one. */
public class WrongLeapYear {
    public static void main(String[] args) {
        int count = 0;
        for (int year = 1901; year <= 2100; year++) {
            if (year % 4 == 0) {
                count++;
            }
        }
        System.out.println("1900 leap? " + (1900 % 4 == 0));
        System.out.println("Leap years from 1901 to 2100: " + count);
    }
}
```

```
1900 leap? true
Leap years from 1901 to 2100: 50
```

“Every fourth year” is the old Julian rule, and it is what most people remember. It is right for every year from 1901 to 2099, which is exactly why the bug survives casual testing, and wrong for 1900 and 2100. A test list that contains 1900, 2000 and 2100 — the three kinds of century year — catches it immediately. **Listing 4.70** is the Python version.

Listing 4.70 `wrong_leap_year.py`: the same Julian rule in Python.

```python
"""Divisible by 4" alone is the Julian rule, not the Gregorian one."""
count = 0
for year in range(1901, 2101):
    if year % 4 == 0:
        count += 1
print(f"1900 leap? {1900 % 4 == 0}")
print(f"Leap years from 1901 to 2100: {count}")
```

```
1900 leap? True
Leap years from 1901 to 2100: 50
```

CHECK YOURSELF. Write a method `dayOfYear()` that returns the number of the day within its year, 1 for 1 January. Use a `for` loop over the months before the date’s month. What is the day of the year of 2024-03-01 and of 2023-12-31?

Answer. `int n = day; for (int m = 1; m < month; m++) { n = n + daysInMonth(year, m); } return n;`. For 2024-03-01: $31 + 29 + 1 = 61$. For 2023-12-31: the eleven months before December have $31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 30 = 334$ days, and $334 + 31 = 365$, as expected in a common year.

4.17 The Problems Side by Side

Fourteen problems can blur into one another. **Table 4.8** puts them in a row with the one idea that each one teaches best and the trap it guards against. If you remember only the right-hand column of this table, you will avoid most of the bugs that beginners write in their first year.

Table 4.8 The fourteen problems of this chapter, the design idea each one teaches, and the trap its tempting wrong solution demonstrates.

Problem	Design idea	Trap shown by the tempting wrong solution
4.1 Fraction calculator	one formula per method	integer division; adding tops and bottoms
4.2 Clock angle	normalize, then round	negative `%` in Java; half-up versus half-to-even
4.3 Computer account	private fields, one object per account	`==` on strings
4.4 Complex numbers	operations return new objects	forgetting $i^2 = -1$
4.5 Weight	store once, derive the rest	a second field that goes stale
4.6 Fraction class	immutability, chaining	changing `this` inside `getSum`
4.7 Player and enemy	each object guards its own fields	writing another object's field
4.8 Rupiah	exact money in `long`	grouping from the left; money in `double`
4.9 Time of day	one internal number, hidden	no carry, no wrap, no padding
4.10 Laundry receipt	composition, `switch`	truncating the weight; locale
4.11 Tidying a name	one private rule used three times	`split(" ")` and empty words
4.12 Password rules	count in the loop, decide after it	returning inside the loop
4.13 Tracing loops	trace tables	`n > 9`; `/` instead of `//`
4.14 Leap years	a rule as one Boolean expression	"divisible by 4" alone

Two patterns stand out. First, every class that represents a *value* — a fraction, a complex number, an amount of money, a time, a date — was made immutable, with operations that return new objects. Second, almost every trap is invisible on the "nice" examples and visible only at an edge: an odd minute, a negative number, a second space, a century year. Testing the edges is not an extra; it *is* testing.

4.18 Profitina in Practice

Profitina (app.profitina.com) measures how ready a brand's website is to be cited by AI answer engines and helps fix what it finds. The problems of this chapter are small, but each of their ideas appears, at full size, in such a product.

Small classes, each tested on its own. The scanner is built from signal classes: each signal — say, whether a page carries structured data — is its own small class with one job, scored and explained independently, exactly as `Rupiah`, `TimeOfDay` and `PasswordChecker` each have one job here. Small classes are easy to test at their edges, and Profitina's release process makes that a rule: the full automated suite, well over a thousand tests, must be green before any version is released. The `PASS`/`FAIL` lines of this chapter's drivers are the same idea in miniature.

Money and exact numbers. Profitina's PROFIT utility coin is quoted against the rupiah, and a coin with six decimal places is, like the rupiah amounts of Problem 4.8, best handled as a whole number of its smallest unit rather than as a floating-point value. Wherever an amount is displayed to an Indonesian user, it is grouped with dots, as `Rupiah.toString` does.

Passwords are never printed. Problem 4.3 had to provide `printPassword` because the problem asked for it. A real product does the opposite. Profitina's self-service account settings, built to respect Indonesia's personal data protection law (UU PDP 27/2022) [12], let a user change the password, but the password itself is stored only as an Argon2id hash, is never shown, and a changed password requires proof of the old one — the *check yourself* of Problem 4.3, taken seriously.

Time and dates. Profitina's historical dashboard shows a weekly score trend per brand and re-scans automatically on a schedule. "Seven days after this scan", "is this scan older than the last one", "which week does this date fall in": these are the questions `TimeOfDay` and `SimpleDate` answer in miniature, and the traps of Problems 4.9 and 4.14 — wrap-around, month ends, leap years — are the ones every scheduler must get right.

All of this remains advice: a score tells a brand owner where to look, and the owner decides what to change. The runnable practice for this book stays with Profitina Laundry, whose receipt of Problem 4.10 combines, in forty lines, the money of Problem 4.8, the conditionals of Chapter 3 and the idea of one object holding another.

4.19 Common Mistakes, and How to Avoid Them

Dividing two integers and expecting a fraction. In Java, `1 / 2` is `0`. Write `1 / 2.0`, or cast one operand to `double`, whenever you want a fractional result; in Python, remember that `/` always gives a floating-point number and `//` is the integer division (Problems 4.1, 4.2, 4.13).

Trusting `%` with negative numbers in Java. `-105 % 360` is `-105` in Java [7] and `255` in Python [5]. In Java, normalize with `(x % m + m) % m` (Problems 4.2 and 4.9).

Comparing strings with `==`. Use `equals` in Java; in Python, `==` compares contents and `is` compares identity (Problem 4.3).

Rounding with the wrong rule. Decide which rule the problem means: half up (`Math.round`), half to even (`Math.rint`, Python's `round`), towards zero (a cast), or up (`Math.ceil`). They differ exactly on the cases that tests should contain (Problems 4.2 and 4.10).

Comparing `double` values with `==`. Computed floating-point results carry tiny errors; compare their distance with a tolerance, or use whole numbers for money (Problems 4.5 and 4.8).

Storing the same fact twice. A second field that can be derived from the first will one day disagree with it; compute it instead (Problem 4.5).

Changing `this` in a method that should return a result. A method called `getSum` must not modify its object. Return a new object (Problems 4.4 and 4.6).

Reaching into another object. Let each object change only its own fields, and let others ask it through methods (Problem 4.7).

Deciding inside a loop what can only be decided after it. "No character is a digit" is known only after the last character (Problem 4.12).

Off-by-one loop conditions. Trace the first and the last pass of every loop on paper; if the table has one row too few or too many, so does the loop (Problem 4.13).

Forgetting `this.` in a constructor. `realName = realName;` assigns a parameter to itself and leaves the field empty (Problem 4.3).

Testing only the examples in the statement. Add zero, negatives, boundaries, an empty string, an odd minute and a century year. Every tempting wrong solution in this chapter passed at least one "nice" example [13].

4.20 Chapter Summary

This chapter consolidated Chapters 1 to 3 by solving problems rather than introducing new features.

A method. Section 4.2 set out eight steps: restate the problem; work examples by hand; find the nouns and verbs; draw the class box; write the tests first; write the code in small pieces; run, compare and debug; and look back for edge cases, tempting wrong solutions and variants. Plain Java test drivers with a `check` helper and Python's `unittest` turned the hand examples into automatic checks; the drivers move to JUnit 6 by replacing each `check` with `assertEquals`.

Seven classic problems. Problem 4.1 added and multiplied fractions with whole numbers, and showed integer division as the first trap. Problem 4.2 computed a clock angle by Equation (4.3) to Equation (4.5), and showed that "rounded" means round half to even there, and that Java's `%` keeps the sign of its left operand. Problem 4.3 built a first real object, with private fields and `equals`. Problem 4.4 made operations return new objects. Problem 4.5 stored one fact and derived the other. Problem 4.6 turned Problem 4.1 into an immutable class whose results can be chained. Problem 4.7 had two classes send each other messages and let each guard its own fields.

Seven new problems. Rupiah amounts as exact whole numbers with digits grouped from the right; a time of day stored as minutes since midnight; a laundry receipt that holds a `Rupiah` total and charges per started kilogram; a name tidied character by character; password rules counted in one loop and decided after it; loops traced by hand in tables, including Euclid's algorithm Equation (4.8); and the Gregorian leap-year rule Equation (4.9) with a date that knows its next day.

Looking ahead. Chapter 5 brings arrays and lists — so that a receipt can hold any number of lines as objects rather than as text — together with keyboard input, exceptions and files. The `StringIndexOutOfBoundsException` of Problem 4.11 will then be something you can catch and handle, rather than only avoid.

4.21 Exercises

4.1 Extend `Q16Fraction` with `getDifference(other)` and `getQuotient(other)`, each returning a new fraction. What do `f2.getDifference(f1)` and `f2.getQuotient(f1)` print for $f_1 = \frac{1}{2}$ and $f_2 = \frac{3}{7}$ (without reducing)?

4.2 Using Equation (4.3) to Equation (4.5), compute by hand the counterclockwise angle at 7:20, 12:30 and 0:01.

4.3 At which times between 3:00 and 3:59 do the two hands of a clock overlap exactly? Give the answer to the nearest second, and say what `Q12Time.angle` returns at 3:16.

4.4 Write a method `equalsFraction(Q16Fraction other)` that returns `true` when two fractions have the same value, such as $\frac{1}{2}$ and $\frac{2}{4}$, without reducing either and without using `double`.

4.5 Trace `Rupiah.toString` for the amount -1000: list the characters read, the count and the result after each pass, and the string returned.

4.6 Give three test cases for `TimeOfDay.plusMinutes` that would each catch a different one of the three bugs of **Listing 4.44**, and state the expected result of each.

4.7 For Budi's express order of Problem 4.10, the washing machine breaks and 1.2 kg of the 4.5 kg are returned unwashed. Compute the new total for 3.3 kg of `wash+iron`, express.

4.8 Write the trace table of `digitSum(90210)`. How many passes does the loop make, and why does the zero in the middle not end the loop?

4.9 Using Equation (4.9), decide for each of 1600, 1700, 2028 and 2400 whether it is a leap year. Then compute by hand how many days lie from 2024-02-27 to 2024-03-02, and check your answer by counting `nextDay` steps.

4.10 A classmate writes `isStrong` as `return password.length() >= 8 && hasDigit(password);`, using the `hasDigit` of **Listing 4.59**. Give one strong password that this version rejects and one weak password that it accepts, under the four rules of Problem 4.12.

4.11 Design (draw the class box, and list the tests) a class `LaundryMachine` with a name, a capacity in kilograms, and a current load. It offers `load(kg)`, which refuses a load that would exceed the capacity, `unload()`, and `freeCapacity()`. Which of this chapter's design ideas do you use?

4.12 Apply the eight steps of Section 4.2 to this problem, writing out each step: "Given a whole number of seconds, print it as hours, minutes and seconds in the form `H:MM:SS`."

4.22 Solutions to the Exercises

4.1 Difference: $\frac{e_1 d_2 - d_1 e_2}{d_1 d_2}$, so `getDifference` returns `new Q16Fraction(numerator * other.getDenominator() - denominator * other.getNumerator(), denominator * other.getDenominator())`. For $\frac{3}{7} - \frac{1}{2}$: $3 \cdot 2 - 7 \cdot 1 = -1$ and $7 \cdot 2 = 14$, so `-1/14`. Quotient: dividing by $\frac{e}{d}$ is multiplying by $\frac{d}{e}$, so `getQuotient` returns `new Q16Fraction(numerator * other.getDenominator(), denominator * other.getNumerator())`. For $\frac{3}{7} \div \frac{1}{2} = \frac{3 \cdot 2}{7 \cdot 1}$ it prints `6/7`. (A quotient by a fraction with numerator 0 is undefined; Chapter 5's exceptions are the proper way to refuse it.)

4.2 7:20: $\theta_{\text{hour}} = 30 \cdot \left(7 + \frac{20}{60}\right) = 210 + 10 = 220$, $\theta_{\text{minute}} = 120$, difference 100, result 100. 12:30: $30 \cdot (0 + 0.5) = 15$, minute hand 180, difference -165, modulo 360 it is 195. 0:01: $30 \cdot \frac{1}{60} = 0.5$, minute hand 6, difference -5.5, modulo 360 it is 354.5, which rounds half to even to 354.

4.3 The hands overlap when $30 \cdot \left(3 + \frac{m}{60}\right) = 6m$, that is $90 + 0.5m = 6m$, so $5.5m = 90$ and $m = \frac{90}{5.5} = 16.\overline{36}$ minutes, which is 16 minutes and $0.\overline{36} \cdot 60 \approx 21.8$ seconds: about 3:16:22. At 3:16 the difference is $90 + 8 - 96 = 2$, so `angle(3, 16)` returns 2; at 3:17 it is $90 + 8.5 - 102 = -3.5$, which normalizes to 356.5 and rounds (half to even) to 356. The hands pass each other between the two minutes.

4.4 Two fractions $\frac{a}{b}$ and $\frac{c}{d}$ with positive denominators are equal exactly when $ad = bc$ (multiply both sides of $\frac{a}{b} = \frac{c}{d}$ by bd). So `return numerator * other.getDenominator() == denominator * other.getNumerator();`. For $\frac{1}{2}$ and $\frac{2}{4}$: $1 \cdot 4 = 4 = 2 \cdot 2$, so `true`. Using `double` would risk rounding errors, as in Problem 4.5.

4.5 `digits` is `"1000"` (the absolute value). Reading from the right: `0`, count 1, result `0`; `0`, count 2, `00`; `0`, count 3, and since `i = 1 > 0` a dot is added: `.000`; `1`, count 4, `1.000`. Because the amount is negative, the method returns `-Rp1.000`.

4.6 (a) `new TimeOfDay(22, 30).plusMinutes(45)` must give `23:15`; the no-carry bug gives `22:75`. (b) `new TimeOfDay(22, 30).plusMinutes(150)` must give `01:00`; the no-wrap bug gives `25:0`. (c) `new TimeOfDay(8, 0).plusMinutes(5)` must give `08:05`; the no-padding bug gives `8:5`. One test per bug is the ideal: when a test fails, its name tells you what broke.

4.7 [3.3] = 4 kg at Rp10.000 is 40000; express adds 20000; the total is 60000, printed `Rp60.000`. (Before, with 5 kg, it was Rp75.000.)

4.8 Passes: digit 0, total 0, $n = 9021$; digit 1, total 1, $n = 902$; digit 2, total 3, $n = 90$; digit 0, total 3, $n = 9$; digit 9, total 12, $n = 0$. Five passes, sum 12. The loop condition tests n, not the digit: a zero *digit* leaves n positive, so the loop continues; only $n = 0$ ends it.

4.9 1600: divisible by 400, leap. 1700: divisible by 100 but not by 400, not leap. 2028: divisible by 4, not by 100, leap. 2400: divisible by 400, leap. From 2024-02-27: `nextDay` gives 02-28, 02-29 (2024 is a leap year), 03-01, 03-02 — four steps, so four days.

4.10 The classmate's version checks only the length and a digit, and its `hasDigit` looks only at the first character. A strong password it rejects: `Laundry7` — the digit comes last, so `hasDigit` returns `false`. A weak password it accepts: `1sesamesesame` — long enough and starting with a digit, but without a capital letter, which the version never checks.

4.11 Class box: fields `- name : String`, `- capacityKg : double`, `- loadKg : double`; methods `+ LaundryMachine(name, capacityKg)`, `+ load(kg : double) : boolean`, `+ unload() : void`, `+ freeCapacity() : double`. `load` returns `false` and changes nothing when `kg <= 0` or `loadKg + kg > capacityKg`; `unload` sets `loadKg` to 0; `freeCapacity` returns `capacityKg - loadKg` — derived, not stored (Problem 4.5). Tests: a 7 kg machine accepts 3 kg and then 4 kg (exactly full, boundary), refuses 0.1 kg more, reports free capacity 0, and after `unload` reports 7. Ideas used: private fields guarded by methods (Problems 4.3 and 4.7), refusing bad input early (Problem 4.10), store once and derive (Problem 4.5), and testing at the boundary (Problem 4.12).

4.12 Step 1: input a whole number $s \geq 0$ of seconds; output a text `H:MM:SS`. Step 2: $3725 = 1 \cdot 3600 + 2 \cdot 60 + 5$, so `1:02:05`; 59 gives `0:00:59`; edge 0 gives `0:00:00`. Step 3: the noun is a *duration* with hours, minutes, seconds; the verb is *format*. Step 4: a class `Duration` with one field `- seconds : int` (one internal number, as in Problem 4.9) and methods `getHours()` $= s/3600$, `getMinutes()` $= (s \bmod 3600)/60$, `getSeconds()` $= s \bmod 60$, and `toString()`. Step 5: tests for 3725, 59, 0 and 86399 (`23:59:59`). Step 6: `toString` returns `String.format("%d:%02d:%02d", getHours(), getMinutes(), getSeconds())`. Step 7: run the tests; a forgotten `%02d` shows up at once as `1:2:5`. Step 8: the tempting wrong solution computes minutes as `s / 60`, which gives 62 for 3725; a variant would print days as well for durations over 24 hours.

References

[1] G. Pólya, *How to Solve It: A New Aspect of Mathematical Method*, 2nd ed. Princeton, NJ, USA: Princeton University Press, 1957.

[2] The JUnit Team, *JUnit User Guide*, version 6.1.3, Aug. 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://docs.junit.org/current/

[3] Python Software Foundation, “unittest — Unit testing framework,” *The Python Standard Library*, Python 3.14 documentation, 2025. [Online]. Available: https://docs.python.org/3/library/unittest.html

[4] *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019, 2019.

[5] Python Software Foundation, “Built-in Functions” and “Expressions,” *The Python Standard Library* and *The Python Language Reference*, Python 3.14 documentation, 2025. [Online]. Available: https://docs.python.org/3/

[6] Oracle, *Java Platform, Standard Edition 21 API Specification* (classes `java.lang.Math`, `java.util.Formatter`, `java.util.Locale`), 2023. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/

[7] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*. Oracle, 2023, §15.17.3 (remainder operator) and §15.21.3 (reference equality).

[8] J. Bloch, *Effective Java*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2018, Item 17: “Minimize mutability.”

[9] P. A. Grassi *et al.*, “Digital Identity Guidelines: Authentication and Lifecycle Management,” NIST Special Publication 800-63B, National Institute of Standards and Technology, Gaithersburg, MD, USA, 2017 (with updates).

[10] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, §4.5.2.

[11] E. M. Reingold and N. Dershowitz, *Calendrical Calculations: The Ultimate Edition*, 4th ed. Cambridge, UK: Cambridge University Press, 2018.

[12] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, 2022.

[13] C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 13th ed. Hoboken, NJ, USA: Pearson, 2024.

Chapter 5

Arrays, Lists, Input, Exceptions, and Files

By the end of this chapter you will be able to keep many values under one name, in a fixed-size array and in a list that grows by itself; draw what an array looks like in memory and explain why its first index is 0 and its last is one less than its length; read what a person types at the keyboard, in Java with `Scanner` and in Python with `input()`; read a stack trace and a traceback calmly, line by line, and find the line that failed; catch an exception, let one travel to a caller that knows what to do with it, clean up with `finally`, and throw one of your own; tell a checked exception from an unchecked one and say why Java makes the difference and Python does not; write your own exception classes; and read and write text files, including CSV files, closing every file you open even when something goes wrong. The chapter ends with Profitina Laundry loading its orders from a CSV file, rejecting every bad line with a precise explanation, and saving the good orders back to disk, in Java and in Python.

5.1 Where We Left Off

Up to now every object we have made has lived alone. A variable held one number, one string, one circle, one laundry order. Chapter 3 gave us classes, objects, strings, types, conditionals and loops, and Chapter 4 put all of that to work on a series of worked problems. Those programs were honest and complete, but they shared a quiet assumption: that we knew in advance exactly how many things we would handle, and that nothing would ever go wrong.

A real laundry does not work like that. On a Monday morning Profitina Laundry might take in four orders or forty. A customer might type “three” where the program expects the number 3. The file that holds last week’s orders might have been moved, renamed, or edited by hand by someone who left a field empty. A program that meets any of these situations with a wall of red text and an abrupt stop is not finished; it is a prototype.

This chapter fills three gaps at once, and they belong together more than it might first appear.

The first gap is **many values**. We learn two containers: the *array*, which is the oldest and simplest, a row of numbered slots of fixed size; and the *list* (`ArrayList` in Java, the built-in `list` in Python), which grows as we add to it.

The second gap is **input**. A program that only prints is a speech, not a conversation. We learn to read what a person types, first from the keyboard and later from a file.

The third gap is **failure**. Once a program reads input it no longer controls its own data, and things will go wrong. Java and Python both answer this with *exceptions*: objects that describe what went wrong, which travel from the place where the problem is detected to the place where someone can do something about it. Exceptions and files are taught together here for a good reason: nearly everything that touches a file can fail, and Java will not even compile file-reading code until you have said what should happen when it does.

We will work in the order that the ideas depend on each other: arrays, then lists, then input, then exceptions, then files. Every example is a complete program, first in Java and then in Python, with its real output.

5.2 Many Values Under One Name: Arrays

5.2.1 The pigeonhole rack

We have already met an array, even if we did not look at it closely. Every Java program we have written starts like this:

```
public class Hello {
    public static void main(String[] args) {
        ...
```

The `String[] args` in the declaration of `main` is an array. The square brackets after `String` say "not one `String`, but a row of them".

A good picture of an array is a pigeonhole rack in a post room: a single piece of furniture with a fixed number of slots, each slot labeled with a number, each able to hold one item. You decide how many slots the rack has when you build it, and after that the number never changes. You can put something in slot 3, take a look at what is in slot 0, or replace the contents of slot 4, but you cannot add a sixth slot to a rack built with five. The idea is that we create an array knowing beforehand how big it needs to be.

> **DEFINITION.** An *array* is a fixed-length sequence of slots, called *elements*, all of the same type, stored one after another and identified by an integer *index*. In Java and in Python the first index is 0, so an array of length n has the indexes $0, 1, \ldots, n-1$.

Figure 5.1 shows an array of five strings as the computer holds it. On the left is the variable `fruitArray`. It does not contain the five fruits itself; it contains a *reference*, the address of the place in memory where the array object lives. The array object is the row of five slots on the right, together with a record of its length. Below each slot is its index.

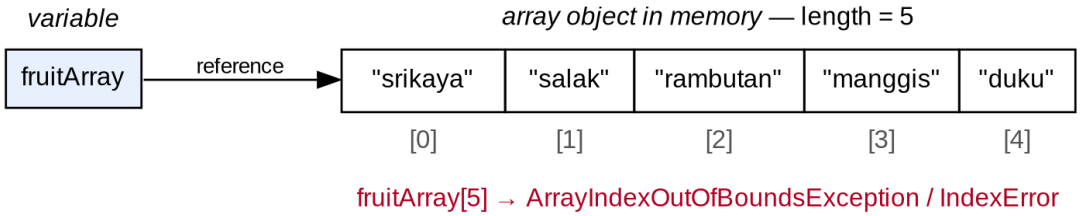


Figure 5.1 An array of five strings in memory. The variable holds a reference to the array object; the slots are numbered from 0 to 4, and index 5 does not exist.

Why start at 0 rather than 1? Because the index is really a distance. The slots of an array sit side by side in memory, each taking the same number of bytes. If the array begins at memory address b and every slot is s bytes wide, then slot i begins at

$$\text{address}(i) = b + i \cdot s \tag{5.1}$$

The first slot is at distance zero from the start, so its index is 0. Equation (5.1) is also why reading any slot of an array takes the same short time no matter how long the array is: the computer does one multiplication and one addition and goes straight to the slot, without walking past the ones before it. That property, called *random access*, is the array's great strength.

The last index is therefore $n - 1$, one less than the length. If you remember only one sentence from this section, make it that one: *an array of length n has no element with index n.*

5.2.2 Creating, filling, and walking an array

Suppose Profitina Laundry's owner is fond of exactly five fruits and wants their names in a program. In Java, we create an array of five `String` slots and fill it one slot at a time. **Listing 5.1** does that, prints the array, and then walks through it with a `for` loop.

Listing 5.1 `FruitArray.java`: creating, filling, and traversing a Java array, plus the shorthand initializer.

```
import java.util.Arrays;

public class FruitArray {
    public static void main(String[] args) {
        String[] fruitArray = new String[5];      // five slots, indexes 0 to 4
        System.out.println("A fresh array: " + Arrays.toString(fruitArray));

        fruitArray[0] = "srikaya";                // we number from zero
        fruitArray[1] = "salak";
        fruitArray[2] = "rambutan";
        fruitArray[3] = "manggis";
        fruitArray[4] = "duku";

        System.out.println("The element at position 0 is " + fruitArray[0]);
        for (int fruitIndex = 0; fruitIndex < fruitArray.length; fruitIndex++) {
            System.out.println("Fruit " + fruitIndex + " is " + fruitArray[fruitIndex]);
        }

        // Shorthand: the size is implied by the number of values in the braces.
        String[] secondFruitArray = {"strawberry", "raspberry", "blackcurrant"};
        System.out.println("The shorthand array has length " + secondFruitArray.length);
        for (String fruit : secondFruitArray) {
            System.out.println("I like " + fruit);
        }

        int[] tenInts = new int[10];              // numbers start as 0
        System.out.println("Ten ints: " + Arrays.toString(tenInts));
    }
}
```

```
A fresh array: [null, null, null, null, null]
The element at position 0 is srikaya
Fruit 0 is srikaya
Fruit 1 is salak
Fruit 2 is rambutan
Fruit 3 is manggis
Fruit 4 is duku
The shorthand array has length 3
I like strawberry
I like raspberry
I like blackcurrant
Ten ints: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Read the listing line by line.

- `String[] fruitArray = new String[5];` does two things. The left side declares a variable whose type is "array of `String`". The right side, `new String[5]`, builds a new array object with five slots in memory and hands back a reference to it. We are putting the string `"srikaya"` in the first position, which is numbered zero.

- A brand-new array is never truly empty: every slot holds a *default value*. For an array of objects such as strings the default is `null`, meaning "no object here yet"; for an array of `int` the default is `0`; for `boolean` it is `false`. The first line of output shows five `null`s, and the last line shows ten zeros. `Arrays.toString` is a helper from `java.util.Arrays` that turns an array into readable text; printing the array directly would show only an unhelpful code such as `[Ljava.lang.String;@1b6d3586`.
- `fruitArray[0] = "srikaya";` puts a value in slot 0. The number in square brackets says which position we want to set or read. Note that it is not possible just to *add* something to an array; we always have to say where it goes.
- `fruitArray.length` is the number of slots. Notice that there are no parentheses: `length` is a field of the array, not a method. (Strings, confusingly, use `length()` with parentheses. The compiler will tell you if you mix them up.)
- The loop `for (int fruitIndex = 0; fruitIndex < fruitArray.length; fruitIndex++)` visits every index from 0 up to, but not including, the length. We could have written `fruitIndex < 5`, and for this array it would work, but it may not always: if the array is later made bigger, a hard-coded 5 silently skips the new slots. We should always use `.length` when traversing an array.
- `String[] secondFruitArray = {"strawberry", "raspberry", "blackcurrant"};` is the *shorthand* form. When we know the contents at the moment of creation, we list them in braces and the compiler counts them: this array has length 3. The two ways are equivalent; writing `new String[3]` and then three assignments would build exactly the same array. It is a useful shortcut, but be sure you understand the long form first.
- `for (String fruit : secondFruitArray)` is the *enhanced for loop*, read aloud as "for each `fruit` in `secondFruitArray`". It visits the elements in order without mentioning an index at all. Use it whenever you need the values but not their positions.
- `int[] tenInts = new int[10];` answers a natural question: an array can hold any type, not only strings. Here are ten `int`s. An array can equally hold `double`s, `boolean`s, `char`s, objects of your own classes, or even other arrays.

Python does not have a fixed-size array built into the language. Its everyday sequence is the `list`, which we study properly in Section 5.3. To imitate a five-slot rack we make a list of five `None`s; `None` plays the role of Java's `null`. **Listing 5.2** is the same program.

Listing 5.2 `fruit_array.py`: the same program with a Python list of fixed length.

```
fruit_array = [None] * 5              # five slots, indexes 0 to 4
print("A fresh list:", fruit_array)

fruit_array[0] = "srikaya"            # we number from zero
fruit_array[1] = "salak"
fruit_array[2] = "rambutan"
fruit_array[3] = "manggis"
fruit_array[4] = "duku"

print("The element at position 0 is " + fruit_array[0])
for fruit_index in range(len(fruit_array)):
    print("Fruit " + str(fruit_index) + " is " + fruit_array[fruit_index])

# Shorthand: the size is implied by the number of values in the brackets.
second_fruit_array = ["strawberry", "raspberry", "blackcurrant"]
print("The shorthand list has length", len(second_fruit_array))
for fruit in second_fruit_array:
    print("I like " + fruit)

ten_ints = [0] * 10                   # numbers do not start as 0 unless we say so
print("Ten ints:", ten_ints)
```

```
A fresh list: [None, None, None, None, None]
The element at position 0 is srikaya
Fruit 0 is srikaya
Fruit 1 is salak
Fruit 2 is rambutan
Fruit 3 is manggis
Fruit 4 is duku
The shorthand list has length 3
I like strawberry
I like raspberry
I like blackcurrant
Ten ints: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The differences are small but worth naming. `[None] * 5` builds a list of five `None`s by repeating a one-element list five times. `len(fruit_array)` plays the part of `fruitArray.length`. The loop `for fruit_index in range(len(fruit_array))` counts from 0 up to but not including the length, exactly like the Java loop; `range(n)` produces 0,1, … , $n - 1$. The shorthand is simply a list written in square brackets. Python's `for fruit in second_fruit_array` is its only kind of `for` loop, and it is the same idea as Java's enhanced `for`. Finally, Python does not fill a new list with zeros on its own: `[0] * 10` says so explicitly.

> **KEY INSIGHT.** In both languages the loop that walks an array has the same shape: start at 0, stop *before* the length. Writing `<` rather than `<=` in that condition is the single most important habit in this section.

(Python also has an `array` module for compact arrays of numbers, and the scientific library NumPy has powerful fixed-size arrays. For ordinary programs, the list is what Python programmers use.)

5.2.3 Stepping off the end

What would happen if we wrote `fruitArray[5] = "durian";`? The array has slots 0 to 4. Slot 5 does not exist. **Listing 5.3** makes this mistake on purpose, in the most common way it happens in practice: a loop condition written with `<=` instead of `<`.

Listing 5.3 `ArrayBounds.java`: a loop that runs one step too far and writes past the end of a five-element array.

```java
import java.util.Arrays;

public class ArrayBounds {
    public static void main(String[] args) {
        String[] words = {"one", "two", "three", "four", "five", "six"};
        String[] stringArray = new String[5];     // only five elements...
        for (int i = 0; i <= 5; i++) {            // ...but i runs up to 5
            stringArray[i] = words[i];
            System.out.println("Added " + words[i] + " at index " + i);
        }
        System.out.println("Done: " + Arrays.toString(stringArray));
    }
}
```

```
Added one at index 0
Added two at index 1
Added three at index 2
Added four at index 3
Added five at index 4
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds
for length 5
    at ArrayBounds.main(ArrayBounds.java:8)
```

The loop runs happily for `i` equal to 0, 1, 2, 3 and 4. When `i` becomes 5 the program tries to store `"six"` in a slot that does not exist, and Java refuses. It does not quietly grow the array, and it does not scribble over whatever memory happens to follow the array (which is what a C program might do, with disastrous results). It stops with an *exception* called `ArrayIndexOutOfBoundsException`, and the message says exactly what was wrong: “Index 5 out of bounds for length 5”. The line beneath, `at ArrayBounds.main(ArrayBounds.java:8)`, names the method and the line number: line 8 is `stringArray[i] = words[i];`. Notice also that `Done:` is never printed. Once the exception is thrown, the rest of `main` is abandoned.

We will learn in Section 5.5 exactly what an exception is and how to catch one. For now, notice the shape of the message and the fact that it points to a precise line.

Listing 5.4 makes the same mistake in Python.

Listing 5.4 `array_bounds.py`: the same off-by-one error in Python.

```python
words = ["one", "two", "three", "four", "five", "six"]
string_array = [None] * 5             # only five elements...
for i in range(6):                    # ...but i runs up to 5
    string_array[i] = words[i]
    print("Added " + words[i] + " at index " + str(i))
print("Done:", string_array)
```

```
Added one at index 0
Added two at index 1
Added three at index 2
Added four at index 3
Added five at index 4
Traceback (most recent call last):
  File "/home/claude/kit/OOP/Code_new/Chapter05/python/array_bounds.py", line 4, in
<module>
    string_array[i] = words[i]
    ~~~~~~~~~~~~^^^
IndexError: list assignment index out of range
```

Python calls the problem an `IndexError`, and its report, called a *traceback*, gives the file, the line number (4), the line itself, and a row of `~` and `^` marks pointing at the part of the line that failed. The path in the `File` line is where the program lived when we ran it; on your machine it will be your own folder.

> **PITFALL.** Python accepts *negative* indexes: `fruit_array[-1]` is the last element and `fruit_array[-5]` the first. That is convenient, but it means that an index computed wrongly as `-1` does not fail in Python; it silently reads the last element. Java has no negative indexes, and `fruitArray[-1]` throws `ArrayIndexOutOfBoundsException`.

5.2.4 Arrays of numbers, and arrays of arrays

Arrays shine when a program must do the same thing to many numbers. Profitina Laundry's first machine washed five loads this morning, and the owner wants the total weight, the heaviest load, and the average. **Listing 5.5** computes all three in one pass, and then uses a *two-dimensional* array to total the number of loads each of three machines washed over three days.

Listing 5.5 `LoadStats.java`: totals, maximum and average of a `double[]`, and row sums of an `int[][]`.

```
public class LoadStats {
    public static void main(String[] args) {
        double[] loadsKg = {4.5, 6.0, 3.2, 7.8, 5.5};

        double total = 0;
        double heaviest = loadsKg[0];
        for (double load : loadsKg) {            // enhanced for: each value in turn
            total += load;
            if (load > heaviest) {
                heaviest = load;
            }
        }
        System.out.printf("%d loads, total %.1f kg, heaviest %.1f kg, average %.2f kg%n",
                loadsKg.length, total, heaviest, total / loadsKg.length);

        // A two-dimensional array: one row per machine, one column per day.
        int[][] washes = {
            {12, 15, 9},
            {8, 11, 14},
            {10, 10, 13}
        };
        for (int m = 0; m < washes.length; m++) {
            int sum = 0;
            for (int d = 0; d < washes[m].length; d++) {
                sum += washes[m][d];
            }
            System.out.println("Machine " + (m + 1) + " washed " + sum + " loads");
        }
        System.out.println("Machine 2 on day 3: " + washes[1][2]);
    }
}
```

```
5 loads, total 27.0 kg, heaviest 7.8 kg, average 5.40 kg
Machine 1 washed 36 loads
Machine 2 washed 33 loads
Machine 3 washed 33 loads
Machine 2 on day 3: 14
```

The first half is a pattern you will write hundreds of times. We keep a running `total`, starting at 0, and a running `heaviest`, starting at the first element (not at 0, because a list of negative numbers would then report 0 as its maximum). The enhanced `for` hands us each `load` in turn. At the end, the average is the total divided by the count: $27.0 / 5 = 5.40$.

The second half introduces an array whose elements are themselves arrays. `int[][] washes` is an array of three rows; each row is an `int[]` of three columns. `washes.length` is the number of rows (3), and `washes[m].length` is the number of columns in row `m` (also 3). The element in row `m` and

column `d` is `washes[m][d]`. Because both indexes start at 0, "machine 2 on day 3" is `washes[1][2]`, which is 14. It is easy to visualize a two-dimensional array as a table, and that ease is one of the reasons arrays remain popular for grids, boards and images.

Listing 5.6 is the same computation in Python, where a two-dimensional array is simply a list of lists.

Listing 5.6 `load_stats.py`: the same statistics with a Python list and a list of lists.

```
loads_kg = [4.5, 6.0, 3.2, 7.8, 5.5]

total = 0
heaviest = loads_kg[0]
for load in loads_kg:                  # each value in turn
    total += load
    if load > heaviest:
        heaviest = load
print(f"{len(loads_kg)} loads, total {total:.1f} kg, heaviest {heaviest:.1f} kg, "
      f"average {total / len(loads_kg):.2f} kg")

# A list of lists: one row per machine, one column per day.
washes = [
    [12, 15, 9],
    [8, 11, 14],
    [10, 10, 13],
]
for m in range(len(washes)):
    total_m = 0
    for d in range(len(washes[m])):
        total_m += washes[m][d]
    print("Machine " + str(m + 1) + " washed " + str(total_m) + " loads")
print("Machine 2 on day 3:", washes[1][2])
```

```
5 loads, total 27.0 kg, heaviest 7.8 kg, average 5.40 kg
Machine 1 washed 36 loads
Machine 2 washed 33 loads
Machine 3 washed 33 loads
Machine 2 on day 3: 14
```

Python's f-string `f"{total:.1f}"` does the job of Java's `printf("%.1f", total)`: one digit after the decimal point. We named the row total `total_m` rather than `sum`, because `sum` is the name of a built-in Python function and hiding it behind a variable is a classic source of puzzling errors later.

5.2.5 An array variable holds a reference

Look again at **Figure 5.1**. The variable holds a *reference*, an arrow to the array, not the slots themselves. This has a consequence that surprises almost every beginner: assigning one array variable to another copies the arrow, not the array. **Listing 5.7** demonstrates it, and then shows three correct ways to make a real copy.

Listing 5.7 `ArrayCopy.java`: aliasing, copying with `Arrays.copyOf` and `System.arraycopy`, and comparing with `==` and `Arrays.equals`.

```java
import java.util.Arrays;

public class ArrayCopy {
    public static void main(String[] args) {
        int[] original = {1, 2, 3};
        int[] alias = original;                    // copies the reference, not the slots
        alias[0] = 99;
        System.out.println("original after alias[0] = 99: " + Arrays.toString(original));

        int[] copy = Arrays.copyOf(original, original.length);
        copy[1] = 42;
        System.out.println("copy:     " + Arrays.toString(copy));
        System.out.println("original: " + Arrays.toString(original));

        int[] bigger = Arrays.copyOf(original, 6);    // "growing" means a new array
        System.out.println("bigger:   " + Arrays.toString(bigger));

        int[] manual = new int[5];
        System.arraycopy(original, 0, manual, 1, 3);  // from, start, to, start, count
        System.out.println("manual:   " + Arrays.toString(manual));

        int[] twin = {99, 2, 3};
        System.out.println("original == twin? " + (original == twin));
        System.out.println("Arrays.equals(original, twin)? " + Arrays.equals(original,
twin));
    }
}
```

```
original after alias[0] = 99: [99, 2, 3]
copy:     [99, 42, 3]
original: [99, 2, 3]
bigger:   [99, 2, 3, 0, 0, 0]
manual:   [0, 99, 2, 3, 0]
original == twin? false
Arrays.equals(original, twin)? true
```

After `int[] alias = original;` there is still only one array, with two names. Changing `alias[0]` changes what `original[0]` shows, because they are the same slot. Two variables that refer to the same object are called *aliases*.

`Arrays.copyOf(original, original.length)` builds a new array and copies the values into it; afterwards `copy` and `original` are independent, so changing `copy[1]` leaves `original` alone.

This also answers a question the fixed size raises: if an array cannot grow, how do we make one bigger? The answer is that we don't; we make a new, bigger array and copy the old contents into it. `Arrays.copyOf(original, 6)` does that in one call and fills the extra slots with zeros. `System.arraycopy(source, sourceStart, target, targetStart, count)` is the lower-level tool that `copyOf` uses: here it copies three values into `manual` starting at index 1. It works, but it is long-winded, and Section 5.3 will show a class that does this bookkeeping for us.

The last two lines compare arrays. `original == twin` asks "are these the same object?", and the answer is `false`, even though they hold the same numbers. `Arrays.equals(original, twin)` asks "do these hold equal values in the same order?", and the answer is `true`. This is the same distinction as `==` versus `equals` for strings, and for the same reason.

Listing 5.8 tells the same story in Python.

Listing 5.8 `array_copy.py`: aliasing and copying Python lists, and `is` versus `==`.

```
original = [1, 2, 3]
alias = original                      # copies the reference, not the slots
alias[0] = 99
print("original after alias[0] = 99:", original)

copy = original.copy()                # or list(original), or original[:]
copy[1] = 42
print("copy:    ", copy)
print("original:", original)

bigger = original + [0] * 3           # a new, longer list
print("bigger:  ", bigger)

manual = [0] * 5
manual[1:4] = original                # slice assignment copies three values
print("manual:  ", manual)

twin = [99, 2, 3]
print("original is twin?", original is twin)
print("original == twin?", original == twin)
```

```
original after alias[0] = 99: [99, 2, 3]
copy:     [99, 42, 3]
original: [99, 2, 3]
bigger:   [99, 2, 3, 0, 0, 0]
manual:   [0, 99, 2, 3, 0]
original is twin? False
original == twin? True
```

Assignment in Python also copies only the reference, so `alias` and `original` are the same list. `original.copy()`, `list(original)` and the slice `original[:]` all make a new list. `original + [0] * 3` builds a new, longer list. Slice assignment, `manual[1:4] = original`, replaces positions 1, 2 and 3 with the three values. And Python's operators are the other way round from Java's: `is` asks "same object?" while `==` compares the contents. So in Python `original == twin` is `True`.

5.2.6 The array in `main`: command-line arguments

We now know enough to answer the question we started with: what is the `String[] args` in `main` for? When you start a Java program from a terminal, any words you type after the class name are collected into that array. If we type

```
java Hello2 one two three
```

then `args` holds the three strings `"one"`, `"two"` and `"three"`. **Listing 5.9** prints them.

Listing 5.9 `Hello2.java`: printing the command-line arguments.

```
public class Hello2 {
    public static void main(String[] args) {
        System.out.println("I received " + args.length + " argument(s).");
        for (int i = 0; i < args.length; i++) {
            System.out.println("args[" + i + "] = " + args[i]);
        }
    }
}
```

```
I received 3 argument(s).
args[0] = one
args[1] = two
args[2] = three
```

Command-line arguments are important whenever a program needs information that varies from run to run even after compilation: the name of a file to open, a date, a mode. We will use them in

the exercises to pass a file name to a program. In Python the same words arrive in the list `sys.argv`, with one difference shown in **Listing 5.10**: `sys.argv[0]` is the name of the script itself, so the real arguments start at index 1. We run it as `python3 hello2.py one two three`.

Listing 5.10 `hello2.py`: printing the command-line arguments from `sys.argv`.

```
import sys

print("I received " + str(len(sys.argv) - 1) + " argument(s).")
for i in range(1, len(sys.argv)):     # sys.argv[0] is the script's own name
    print("sys.argv[" + str(i) + "] = " + sys.argv[i])
```

```
I received 3 argument(s).
sys.argv[1] = one
sys.argv[2] = two
sys.argv[3] = three
```

In VS Code you can supply arguments without a terminal by adding an `"args"` entry to the run configuration in `launch.json`, as Chapter 2 showed for the debugger.

5.3 Lists That Grow: `ArrayList` and Python's `list`

5.3.1 Why a fixed size is not enough

It is very common in programming to want to store more than one of something without knowing how many. A laundry does not know on Monday morning how many orders will arrive by Monday evening. A program that asks a user to type fruit names until they type `#stop#` cannot know how many names are coming. How many `String` variables would we need? How big should the array be? Any number we choose is wrong: too small and we crash, too large and we waste memory and must keep a separate count of how many slots are really in use.

What we want is a row of pigeonholes to which we can add extra racks when it is full. Java provides exactly that as a class, `java.util.ArrayList`. Python builds it into the language as the `list` type we have already been using.

> **DEFINITION.** A *list* is an ordered sequence of elements whose length can change: elements can be appended, inserted, replaced and removed, and every element still has an integer index starting at 0. Java's `ArrayList<T>` holds elements of type `T`; Python's `list` may hold any objects.

The `<String>` in `ArrayList<String>` is a *type parameter*. Read it for now as "an `ArrayList` that holds `String`s, and only `String`s". If we try to `add` an `Integer` to it, the compiler refuses. Chapter 9 explains these *generic* types in full.

5.3.2 The everyday operations

Listing 5.11 exercises the operations you will use every day: add, size, get, contains, indexOf, set, and the two kinds of remove.

Listing 5.11 `FruitList.java`: the everyday `ArrayList` operations.

```
import java.util.ArrayList;

public class FruitList {
    public static void main(String[] args) {
        ArrayList<String> fruit = new ArrayList<>();   // starts empty
        fruit.add("mango");
        fruit.add("banana");
        fruit.add("peach");
        fruit.add("plum");
        fruit.add("durian");
        fruit.add("plum");
        System.out.println("List: " + fruit + ", size " + fruit.size());
        System.out.println("First fruit: " + fruit.get(0));

        boolean peachIsIn = fruit.contains("peach");
        if (peachIsIn) {
            System.out.println("You entered peach, somewhere, apparently");
        } else {
            System.out.println("No peaches");
        }
        int positionOfPeach = fruit.indexOf("peach");
        int positionOfKiwi = fruit.indexOf("kiwi");
        System.out.println("peach at " + positionOfPeach + ", kiwi at " + positionOfKiwi);

        fruit.set(3, "lemon");                          // plum at 3 becomes lemon
        System.out.println("After set(3, \"lemon\"): " + fruit);
        String removed = fruit.remove(3);               // by position: returns the element
        System.out.println("remove(3) returned " + removed + ": " + fruit);
        boolean gone = fruit.remove("plum");            // by value: returns true or false
        System.out.println("remove(\"plum\") returned " + gone + ": " + fruit);
        System.out.println("remove(\"plum\") again returned " + fruit.remove("plum"));

        for (int j = 0; j < fruit.size(); j++) {
            System.out.println(j + ": " + fruit.get(j));
        }
    }
}
```

```
List: [mango, banana, peach, plum, durian, plum], size 6
First fruit: mango
You entered peach, somewhere, apparently
peach at 2, kiwi at -1
After set(3, "lemon"): [mango, banana, peach, lemon, durian, plum]
remove(3) returned lemon: [mango, banana, peach, durian, plum]
remove("plum") returned true: [mango, banana, peach, durian]
remove("plum") again returned false
0: mango
1: banana
2: peach
3: durian
```

Line by line:

- `import java.util.ArrayList;` is needed at the top because `ArrayList`, unlike an array, is a class from the standard library rather than part of the language.
- `new ArrayList<>()` creates an empty list. The empty angle brackets, nicknamed the *diamond*, tell Java to work out the type parameter from the left-hand side.
- `fruit.add("mango")` appends to the end. The list expands by itself when it gets full, without us having to do anything. Printing a list directly gives readable output, unlike an array.
- `fruit.size()` is the number of elements. Note the three different spellings for "how many": `array.length` (a field), `string.length()` (a method), and `list.size()` (a method).
- `fruit.get(0)` reads the element at a position; the most obvious operation of all.

- `fruit.contains("peach")` answers a yes-or-no question, which is exactly what a `boolean` is for. The listing stores the answer in `peachIsIn` and branches on it.
- `fruit.indexOf("peach")` tells us *where* the peach is: position 2. If the string we look for is not in the list, `indexOf` returns −1, which can never be a valid index, so it is a safe signal for "absent".
- `fruit.set(3, "lemon")` replaces the element at position 3. We have gone off plums of late and prefer lemons.
- The two `remove` calls are very different. `fruit.remove(3)` removes the element at *position* 3, whatever it is, and returns that element (here `"lemon"`). `fruit.remove("plum")` removes the first element *equal to* `"plum"` and returns a `boolean`: `true` if it found one, `false` if not. It returns a `boolean` because the caller may need to know whether anything happened, and the second call shows `false`. Everything after a removed element shifts one place to the left, so indexes change.
- The final loop prints every element with its index, from 0 to `size() - 1`. The loop stops when `j` equals the size.

Python's `list` offers the same operations, sometimes with different names and, importantly, different reactions to absent values. **Listing 5.12** is the same sequence of steps.

Listing 5.12 `fruit_list.py`: the same operations on a Python list.

```
fruit = []                                  # starts empty
fruit.append("mango")
fruit.append("banana")
fruit.append("peach")
fruit.append("plum")
fruit.append("durian")
fruit.append("plum")
print("List:", fruit, "size", len(fruit))
print("First fruit:", fruit[0])

peach_is_in = "peach" in fruit
if peach_is_in:
    print("You entered peach, somewhere, apparently")
else:
    print("No peaches")
position_of_peach = fruit.index("peach")
position_of_kiwi = fruit.index("kiwi") if "kiwi" in fruit else -1   # index() raises if
absent
print("peach at", position_of_peach, "kiwi at", position_of_kiwi)

fruit[3] = "lemon"                          # plum at 3 becomes lemon
print('After fruit[3] = "lemon":', fruit)
removed = fruit.pop(3)                      # by position: returns the element
print("pop(3) returned", removed + ":", fruit)
fruit.remove("plum")                        # by value: returns None, raises if absent
print('remove("plum"):', fruit)
try:
    fruit.remove("plum")
except ValueError as e:
    print('remove("plum") again raised ValueError:', e)

for j in range(len(fruit)):
    print(str(j) + ": " + fruit[j])
```

```
List: ['mango', 'banana', 'peach', 'plum', 'durian', 'plum'] size 6
First fruit: mango
You entered peach, somewhere, apparently
peach at 2 kiwi at -1
After fruit[3] = "lemon": ['mango', 'banana', 'peach', 'lemon', 'durian', 'plum']
pop(3) returned lemon: ['mango', 'banana', 'peach', 'durian', 'plum']
remove("plum"): ['mango', 'banana', 'peach', 'durian']
remove("plum") again raised ValueError: list.remove(x): x not in list
0: mango
1: banana
2: peach
3: durian
```

Three differences deserve attention. First, `"peach" in fruit` is Python's `contains`, and it reads like English. Second, `fruit.index("kiwi")` does *not* return −1 for a missing value; it raises a `ValueError`. That is why the listing checks with `in` first. Third, `fruit.remove("plum")` returns nothing and raises `ValueError` if the value is absent; the listing catches that error with `try` and `except`, a construction we study properly in Section 5.5. Removing by position is `fruit.pop(3)`, which returns the removed element, or `del fruit[3]`, which does not.

Table 5.1 collects the operations side by side. Keep it nearby for the rest of the book.

Table 5.1 The everyday operations on a Java array, a Java `ArrayList`, and a Python `list`.

Operation	Java array `a`	Java `ArrayList<T> list`	Python `list`
Create empty / sized	`new T[n]`	`new ArrayList<>()`	`[]` or `[None] * n`
Create with values	`{x, y, z}`	`new ArrayList<>(List.of(x, y, z))`	`[x, y, z]`
Number of elements	`a.length`	`list.size()`	`len(lst)`
Read position `i`	`a[i]`	`list.get(i)`	`lst[i]`
Replace position `i`	`a[i] = v`	`list.set(i, v)`	`lst[i] = v`
Append	impossible	`list.add(v)`	`lst.append(v)`
Insert at `i`	impossible	`list.add(i, v)`	`lst.insert(i, v)`
Remove position `i`	impossible	`list.remove(i)` (returns it)	`lst.pop(i)` or `del lst[i]`
Remove value `v`	impossible	`list.remove(v)` (returns `boolean`)	`lst.remove(v)` (raises if absent)
Is `v` present?	loop	`list.contains(v)`	`v in lst`
Position of `v`	loop	`list.indexOf(v)` (−1 if absent)	`lst.index(v)` (raises if absent)
Bad index	`ArrayIndexOutOfBoundsException`	`IndexOutOfBoundsException`	`IndexError`

5.3.3 How a list grows

A list that "grows by itself" sounds like magic, but it is the trick from Section 5.2.5 done for us. Inside every `ArrayList` there is an ordinary array, usually with some spare slots at the end. The

list keeps two numbers: its *size*, the number of slots in use, and its *capacity*, the length of the hidden array. As long as size is less than capacity, `add` simply fills the next spare slot. When the array is full, `add` creates a bigger array, copies every reference across, and then adds the new element. **Figure 5.2** shows that moment.

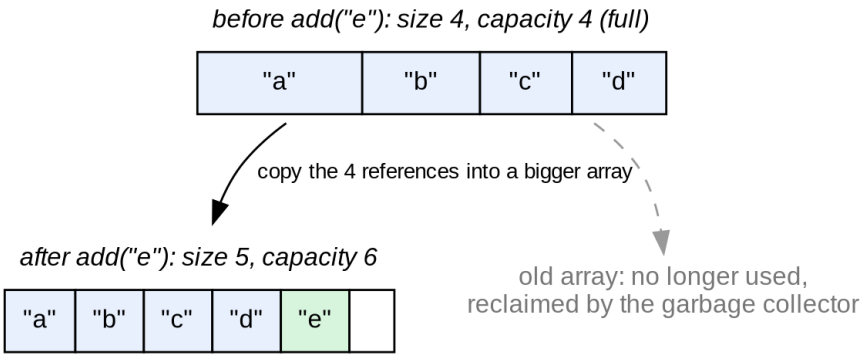


Figure 5.2 What `add` does when the hidden array is full: a larger array is made, the old references are copied into it, and the old array is thrown away.

How much bigger should the new array be? This choice matters more than it seems. Suppose we grew by just one slot each time. Adding the n-th element would then copy the $n-1$ elements already there, and adding n elements to an empty list would copy

$$0 + 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} \tag{5.2}$$

elements in total. For $n = 1000$ that is $1000 \times 999/2 = 499{,}500$ copies; the work grows with the square of n. Now suppose instead that we double the capacity each time the array is full, starting at 1. The copies happen when the size reaches 1,2,4,8, ..., and each copies that many elements. If the largest such power of two below n is 2^k, the total is

$$1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n \tag{5.3}$$

copies. For $n = 1000$, with $2^9 = 512$, that is 1023 copies: fewer than one copy per element on average, even though some individual `add` calls copy hundreds. That is why appending to a list is, on average, as cheap as writing into an array. Java's `ArrayList` grows by half its current capacity each time (a factor of 1.5) and CPython's `list` by a somewhat smaller factor; both are geometric growth, and the argument of Equation (5.3) works for any growth factor greater than 1, only with a different constant. Exercise 5.4 asks you to measure both strategies, and its solution confirms the numbers 499,500 and 1023 by running the code.

5.3.4 Lists of numbers: boxing

Java's type parameter must be a *class* type, never a primitive type. `ArrayList<int>` does not compile. For a list of whole numbers we write `ArrayList<Integer>`, where `Integer` is the *wrapper class* for `int` (there is also `Double` for `double`, `Boolean` for `boolean`, and so on). Java converts automatically between `int` and `Integer`; the conversion to the object is called *boxing* and the conversion back *unboxing*. Mostly this is invisible and pleasant. There is one famous exception, shown in **Listing 5.13**.

Listing 5.13 `BoxingDemo.java`: boxing, unboxing, and the two meanings of `remove` on a list of `Integer`.

```java
import java.util.ArrayList;

public class BoxingDemo {
    public static void main(String[] args) {
        ArrayList<Integer> weights = new ArrayList<>();  // Integer, never int
        weights.add(5);                                  // int 5 is boxed to Integer
        weights.add(10);
        weights.add(3);
        weights.add(7);
        System.out.println("weights: " + weights);

        weights.remove(3);                               // an int: removes INDEX 3
        System.out.println("after remove(3): " + weights);

        weights.remove(Integer.valueOf(3));              // an Integer: removes VALUE 3
        System.out.println("after remove(Integer.valueOf(3)): " + weights);

        int first = weights.get(0);                      // unboxed back to int
        System.out.println("first * 2 = " + (first * 2));
    }
}
```

```
weights: [5, 10, 3, 7]
after remove(3): [5, 10, 3]
after remove(Integer.valueOf(3)): [5, 10]
first * 2 = 10
```

`ArrayList` has two `remove` methods: `remove(int index)` and `remove(Object value)`. When we write `weights.remove(3)`, the argument `3` is an `int`, which matches the first method exactly, so Java removes *index* 3, the value 7, and not the value 3 as we might have hoped. To remove the value 3 we must hand over an `Integer` object: `weights.remove(Integer.valueOf(3))`. This trap bites only lists of `Integer`, and it bites everybody once.

Python has no primitive types and no boxing: an `int` is already an object and may go straight into a list. Python avoids the ambiguity by using different words, as **Listing 5.14** shows: `del weights[3]` for a position, `weights.remove(3)` for a value.

Listing 5.14 `boxing_demo.py`: removing by position and by value in Python.

```python
weights = []                                # any object may go in; ints are objects
weights.append(5)
weights.append(10)
weights.append(3)
weights.append(7)
print("weights:", weights)

del weights[3]                              # by INDEX: removes position 3
print("after del weights[3]:", weights)

weights.remove(3)                           # by VALUE: removes the first 3
print("after weights.remove(3):", weights)

first = weights[0]
print("first * 2 =", first * 2)
```

```
weights: [5, 10, 3, 7]
after del weights[3]: [5, 10, 3]
after weights.remove(3): [5, 10]
first * 2 = 10
```

5.3.5 Array or list?

Good question. Arrays are, we might say, more basic than lists. They need no import, they work with primitive types directly (an `int[]` stores the numbers themselves, not `Integer` objects, so it is smaller and faster), their length is fixed (which is exactly right for things whose number really is fixed, like the seven days of a week or the 8×8 squares of a chessboard), and a two-dimensional array is easy to picture and index. But an array cannot change its length. If you need to make one bigger you are stuck with the long-winded copying of Section 5.2.5, which is exactly what `ArrayList` does for you. **Table 5.2** sums up the choice.

Table 5.2 Choosing between an array and a list.

Question	Array	List (`ArrayList` / `list`)
Is the number of elements known when the container is made, and fixed?	yes: a natural fit	works, but the flexibility is unused
Will elements be added or removed later?	awkward: copy into a new array	yes: `add`, `remove`, `append`, `pop`
Primitive numbers in Java?	`int[]`, `double[]` hold them directly	`ArrayList<Integer>` boxes each one
A grid or table of fixed shape?	`int[][]` is clear and fast	a list of lists works too
Needs `main(String[] args)` or an API that takes an array?	required	convert with `toArray`

In everyday Java, when in doubt, choose `ArrayList`. In everyday Python the question hardly arises: use `list`.

5.4 Reading Input from the Keyboard

5.4.1 `Scanner` and `input()`

By now we know how to make a program print. Quite often, though, we also need it to *read*. Java's simplest tool for reading typed input is the class `java.util.Scanner`. We build one around `System.in`, the program's *standard input*, which is normally the keyboard:

```
Scanner sc = new Scanner(System.in);
String aLineOfText = sc.nextLine();
```

`sc.nextLine()` waits until the user presses Enter and then returns everything typed on that line, without the line break, as a `String`. `Scanner` also offers `nextInt()`, `nextDouble()`, `nextBoolean()` and a plain `next()`, each of which reads one *token*, a run of characters separated from its neighbors by spaces or line breaks, and converts it to the requested type. **Listing 5.15** uses three of them.

Listing 5.15 `SquareInput.java`: reading a line, an integer, a decimal number and another line with `Scanner`.

```java
import java.util.Scanner;

public class SquareInput {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("What is your name?");
        String name = sc.nextLine();                  // the whole line
        System.out.println("Pick an integer: ");
        int value = sc.nextInt();                     // one token, read as an int
        System.out.println("Hello, " + name + ". You picked " + value
                + ". The square of that is " + (value * value) + ".");
        System.out.println("How many kilograms of laundry?");
        double kg = sc.nextDouble();
        sc.nextLine();                                // swallow the rest of that line
        System.out.println("Any note for the courier?");
        String note = sc.nextLine();
        System.out.println(kg + " kg noted, with the note \"" + note + "\".");
    }
}
```

To make every run of the program repeatable, and to show you exactly what was typed, we keep the typed input in a file next to the program, `SquareInput.input.txt`:

```
Sari
12
3.5
Please ring twice
```

and we *redirect* the program's standard input from that file, so that `Scanner` reads the file instead of the keyboard:

```
java SquareInput < SquareInput.input.txt
```

The `<` works the same way in the Windows Command Prompt, in PowerShell (as `Get-Content SquareInput.input.txt | java SquareInput`), and in the macOS and Linux terminals. Because the input now comes from a file, the typed text is not echoed on the screen, so the output below shows only what the program printed:

```
What is your name?
Pick an integer:
Hello, Sari. You picked 12. The square of that is 144.
How many kilograms of laundry?
Any note for the courier?
3.5 kg noted, with the note "Please ring twice".
```

If you run the program without redirection, it waits after each question for you to type.

One line in the listing needs explaining: `sc.nextLine();` with its result thrown away. `nextDouble()` reads the token `3.5` and stops *right after it*, leaving the line break that ended that line unread. If we then called `nextLine()` to read the courier note, it would find that leftover line break at once and return the empty string `""`, and the real note would be read by nobody. The extra `nextLine()` swallows the remainder of the number's line. Every Java programmer is caught by this once. The rule is: *after `nextInt()`, `nextDouble()` or `next()`, call `nextLine()` once before you want a whole line.*

Python's equivalent is the built-in function `input()`, which reads one whole line and returns it as a string, without the line break. There are no typed variants; instead, you convert the string yourself with `int(...)` or `float(...)`. **Listing 5.16** is the same program, run with `python3 square_input.py < square_input.input.txt` and the same input file.

Listing 5.16 `square_input.py`: the same dialogue with `input()`, `int()` and `float()`.

```
print("What is your name?")
name = input()                                  # the whole line, as a str
print("Pick an integer: ")
value = int(input())                            # convert the text to an int
print("Hello, " + name + ". You picked " + str(value)
      + ". The square of that is " + str(value * value) + ".")
print("How many kilograms of laundry?")
kg = float(input())
print("Any note for the courier?")
note = input()                                  # nothing to swallow: input() reads whole
lines
print(str(kg) + ' kg noted, with the note "' + note + '".')
```

```
What is your name?
Pick an integer:
Hello, Sari. You picked 12. The square of that is 144.
How many kilograms of laundry?
Any note for the courier?
3.5 kg noted, with the note "Please ring twice".
```

Because `input()` always consumes a whole line, Python has no leftover-line-break trap. You can also pass the question to `input` directly, as in `name = input("What is your name? ")`, which prints the prompt without moving to a new line. We used separate `print` calls so that the Java and Python outputs match line for line.

What if the user types something that is not a number, say `twelve`? Java's `nextInt()` and Python's `int()` both refuse, and the program crashes. Don't worry: that is normal, and Section 5.5 is devoted to dealing with such errors in a better way.

5.4.2 Reading until a sentinel

When a program cannot know how many values are coming, a common convention is to agree on a special value, called a *sentinel*, that means "no more". **Listing 5.17** reads fruit names until the user types `#stop#`, collecting them in an `ArrayList`. It brings together everything in this chapter so far: input, a `while` loop, and a list that grows.

Listing 5.17 `FruitCollector.java`: reading lines into an `ArrayList` until the sentinel `#stop#`.

```
import java.util.ArrayList;
import java.util.Scanner;

public class FruitCollector {
    public static void main(String[] args) {
        System.out.println("Enter some fruit names. Enter #stop# to finish");
        Scanner s_in = new Scanner(System.in);
        ArrayList<String> fruit = new ArrayList<String>();
        String line = s_in.nextLine();
        // Keep adding fruit to the list
        while (!line.equals("#stop#")) {
            System.out.println("You just typed " + line + ". Adding it!");
            fruit.add(line);
            line = s_in.nextLine();
        }
        System.out.println("You entered " + fruit.size()
                + " fruit. Can't you do better?");
    }
}
```

With this input file:

```
srikaya
salak
rambutan
#stop#
```

the program prints:

```
Enter some fruit names. Enter #stop# to finish
You just typed srikaya. Adding it!
You just typed salak. Adding it!
You just typed rambutan. Adding it!
You entered 3 fruit. Can't you do better?
```

Have a look at what is going on. We set up the `Scanner`, create an `ArrayList` of `String`s, read the first line, and then loop: while the line is not the sentinel, report it, add it to the list, and read the next line. At the end `size()` tells us how many were entered. Notice the *priming read* before the loop: the first `nextLine()` happens before the `while` test, and each later one at the bottom of the loop body, so that the test always examines a line we have just read. Notice also that we compare strings with `equals`, never with `==`, as Chapter 3 explained.

Listing 5.18 is the Python version.

Listing 5.18 `fruit_collector.py`: the same sentinel loop in Python.

```
print("Enter some fruit names. Enter #stop# to finish")
fruit = []
line = input()
# Keep adding fruit to the list
while line != "#stop#":
    print("You just typed " + line + ". Adding it!")
    fruit.append(line)
    line = input()
print("You entered " + str(len(fruit)) + " fruit. Can't you do better?")
```

```
Enter some fruit names. Enter #stop# to finish
You just typed srikaya. Adding it!
You just typed salak. Adding it!
You just typed rambutan. Adding it!
You entered 3 fruit. Can't you do better?
```

One weakness is shared by both versions: if the input ends without a `#stop#` line (for example, the file was cut short, or the user pressed Ctrl+D on macOS or Linux, or Ctrl+Z then Enter on Windows), then `nextLine()` throws `NoSuchElementException` and `input()` raises `EOFError`. Section 5.5 shows how to react to that, and Section 5.8 shows loops that stop cleanly at the end of input without any sentinel at all.

5.5 When Things Go Wrong: Exceptions

5.5.1 A program meets a user

One of the problems with programming is users. We never know when errors will occur; the best we can do is guess what might cause them and soften their effects. Sometimes, to be fair, errors happen because we, the programmers, did not anticipate a problem at all. To cover all the cases, Java and Python give us one uniform way of saying that something has gone wrong: the *exception*.

> **DEFINITION.** An *exception* is an object that describes an abnormal event during the execution of a program. When the event is detected the exception is *thrown* (Python says *raised*): normal execution stops at that point, and the exception travels back through the methods that were

> called until some code *catches* it and handles it. If nothing catches it, the program ends and prints a report.

We have already met three exceptions: `ArrayIndexOutOfBoundsException` and `IndexError` in Section 5.2.3, and `ValueError` from a Python list in Section 5.3.2. Others you will meet very soon are `InputMismatchException` and `NullPointerException`. They are the language's way of saying "something is wrong, but it is the kind of thing a program could do something about".

Look at **Listing 5.19**. What could go wrong with it?

Listing 5.19 `InputTest.java`: reading integers until 99, with no protection against bad input.

```
import java.util.*;

public class InputTest {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        ArrayList<Integer> intList = new ArrayList<>();
        System.out.println("Enter some numbers!");
        int currentVal = s.nextInt();
        while (currentVal != 99) {
            System.out.println("Adding " + currentVal);
            intList.add(currentVal);
            currentVal = s.nextInt();
        }
        System.out.println("Exiting: List is " + intList.toString());
    }
}
```

If you noticed that the danger lies in the user's input, well done. Consider what `int currentVal = s.nextInt()` expects: the user should enter a number. If the user enters a string that cannot be read as a number, such as `b`, Java won't be happy. Here is the input file,

```
1
2
b
3
99
```

and here is what happens:

```
Enter some numbers!
Adding 1
Adding 2
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:947)
    at java.base/java.util.Scanner.next(Scanner.java:1602)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2267)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2221)
    at InputTest.main(InputTest.java:12)
```

Ouch. What does this mean? It means that we need to fix something, and the report tells us what and where.

5.5.2 Reading a stack trace

The first line of the report names the exception: we can safely ignore the words `in thread "main"` and look at the type, `java.util.InputMismatchException`. That makes sense: `nextInt` expects an `int`, and we typed something that cannot be interpreted as one. What follows is called a *stack trace*. It lists every method that was in the middle of running when the exception occurred, beginning with the most recent. The top four lines are inside `Scanner` itself, in the Java library, where the problem was detected; the file names and line numbers there (`Scanner.java:947` and

so on) belong to the library and may differ between Java versions. The last line is the one that matters to us: `at InputTest.main(InputTest.java:12)`. Our class caused the error at line 12, which is the *second* `s.nextInt()`, the one inside the loop. The first `nextInt()` read `1` successfully, and so did the one that read `2`; the third call, reading `b`, failed.

> **KEY INSIGHT.** Read a Java stack trace from the top to learn *what* went wrong, then scan downward for the first line that names *your own* class to learn *where*. Lines that begin with `java.base/` are the library; the problem is almost always in the first line of yours beneath them.

Listing 5.20 is the same program in Python, with the same input.

Listing 5.20 `input_test.py`: the same unprotected loop in Python.

```
int_list = []
print("Enter some numbers!")
current_val = int(input())
while current_val != 99:
    print("Adding " + str(current_val))
    int_list.append(current_val)
    current_val = int(input())
print("Exiting: List is " + str(int_list))
```

```
Enter some numbers!
Adding 1
Adding 2
Traceback (most recent call last):
  File "/home/claude/kit/OOP/Code_new/Chapter05/python/input_test.py", line 7, in <module>
    current_val = int(input())
                  ^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'b'
```

Python calls its report a *traceback*, and it runs in the opposite order: the most recent call is at the *bottom*, which is why the first line says `most recent call last`. So a Python traceback is read from the bottom up: the last line gives the exception type, `ValueError`, and a message that even quotes the offending text, `'b'`; the lines above it give the file, the line number (7) and the code. It is line 7, the `int(input())` inside the loop, that failed.

Stack traces become more interesting when methods call other methods. **Listing 5.21** reads three short order lines. `main` calls `loadAll`, which calls `readOrder` for each line, which calls `parseWeight`. When `parseWeight` meets the weight `-2` it *throws* an exception itself, with the keyword `throw` that Section 5.5.7 explains. The exception is caught in `main`, which prints a message and then the stack trace with `printStackTrace`.

Listing 5.21 `PropagationDemo.java`: an exception thrown three calls deep and caught in `main`.

```java
public class PropagationDemo {
    static double parseWeight(String text) {
        double kg = Double.parseDouble(text);          // may throw NumberFormatException
        if (kg <= 0) {
            throw new IllegalArgumentException("weight must be positive: " + text);
        }
        return kg;
    }

    static void readOrder(String line) {
        String[] parts = line.split(",");
        double kg = parseWeight(parts[1]);
        System.out.println("Read " + parts[0] + ", " + kg + " kg");
    }

    static void loadAll(String[] lines) {
        for (String line : lines) {
            readOrder(line);
        }
        System.out.println("All lines loaded.");       // never reached below
    }

    public static void main(String[] args) {
        String[] lines = {"Sari,3.5", "Budi,-2", "Tono,4"};
        try {
            loadAll(lines);
        } catch (IllegalArgumentException e) {
            System.out.println("Stopped: " + e.getMessage());
            e.printStackTrace(System.out);             // the trail the exception left
        }
    }
}
```

```
Read Sari, 3.5 kg
Stopped: weight must be positive: -2
java.lang.IllegalArgumentException: weight must be positive: -2
    at PropagationDemo.parseWeight(PropagationDemo.java:5)
    at PropagationDemo.readOrder(PropagationDemo.java:12)
    at PropagationDemo.loadAll(PropagationDemo.java:18)
    at PropagationDemo.main(PropagationDemo.java:26)
```

Figure 5.3 shows what happened. The calls went down: `main` to `loadAll` to `readOrder` to `parseWeight`. The exception came back up the same path. `parseWeight` did not reach its `return`; `readOrder` did not reach its `println`; `loadAll` did not go on to the third line, `"Tono,4"`, and never printed “All lines loaded.”; each method was abandoned at the point where it had made its call. Only when the exception reached the `try` in `main`, whose `catch` clause names `IllegalArgumentException`, did it stop. This journey back up the chain of calls is called *propagation*, and the stack trace is the exception’s diary of the journey: line 5 in `parseWeight`, called from line 12 in `readOrder`, called from line 18 in `loadAll`, called from line 26 in `main`.

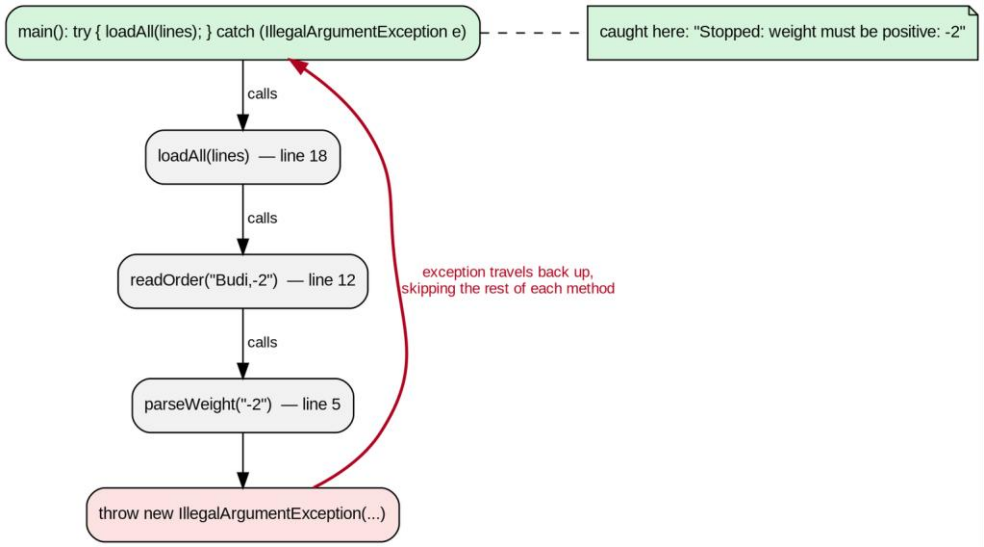


Figure 5.3 Propagation. The calls go down from `main`; the exception travels back up, abandoning each method on the way, until it reaches a matching `catch`.

The Python version, **Listing 5.22**, behaves identically. `traceback.print_exc` prints the traceback of the exception being handled, most recent call last.

Listing 5.22 `propagation_demo.py`: the same propagation in Python.

```
import sys
import traceback


def parse_weight(text):
    kg = float(text)                          # may raise ValueError
    if kg <= 0:
        raise ValueError("weight must be positive: " + text)
    return kg


def read_order(line):
    parts = line.split(",")
    kg = parse_weight(parts[1])
    print("Read " + parts[0] + ", " + str(kg) + " kg")


def load_all(lines):
    for line in lines:
        read_order(line)
    print("All lines loaded.")                # never reached below


def main():
    lines = ["Sari,3.5", "Budi,-2", "Tono,4"]
    try:
        load_all(lines)
    except ValueError as e:
        print("Stopped: " + str(e))
        traceback.print_exc(file=sys.stdout)  # the trail the exception left


if __name__ == "__main__":
    main()
```

```
Read Sari, 3.5 kg
Stopped: weight must be positive: -2
Traceback (most recent call last):
  File "/home/claude/kit/OOP/Code_new/Chapter05/python/propagation_demo.py", line 27, in
main
    load_all(lines)
  File "/home/claude/kit/OOP/Code_new/Chapter05/python/propagation_demo.py", line 20, in
load_all
    read_order(line)
  File "/home/claude/kit/OOP/Code_new/Chapter05/python/propagation_demo.py", line 14, in
read_order
    kg = parse_weight(parts[1])
         ^^^^^^^^^^^^^^^^^^^^^^
  File "/home/claude/kit/OOP/Code_new/Chapter05/python/propagation_demo.py", line 8, in
parse_weight
    raise ValueError("weight must be positive: " + text)
ValueError: weight must be positive: -2
```

5.5.3 A family tree of failures

There are hundreds of exception types. They are not an unordered heap: they form a family tree of classes, in which a more specific exception *extends* a more general one. (The word *extends* is the Java keyword for inheritance, which Chapter 7 studies in depth; for now read "A extends B" as "every A is a kind of B".) Here is the ancestry of `ArrayIndexOutOfBoundsException` in the Java documentation:

```
java.lang.Object
    extended by java.lang.Throwable
        extended by java.lang.Exception
            extended by java.lang.RuntimeException
                extended by java.lang.IndexOutOfBoundsException
                    extended by java.lang.ArrayIndexOutOfBoundsException
```

So an `ArrayIndexOutOfBoundsException` *is an* `IndexOutOfBoundsException`, which *is a* `RuntimeException`, which *is an* `Exception`, which *is a* `Throwable`. Everything that can be thrown in Java extends `Throwable`. **Figure 5.4** shows the part of the tree we meet in this chapter.

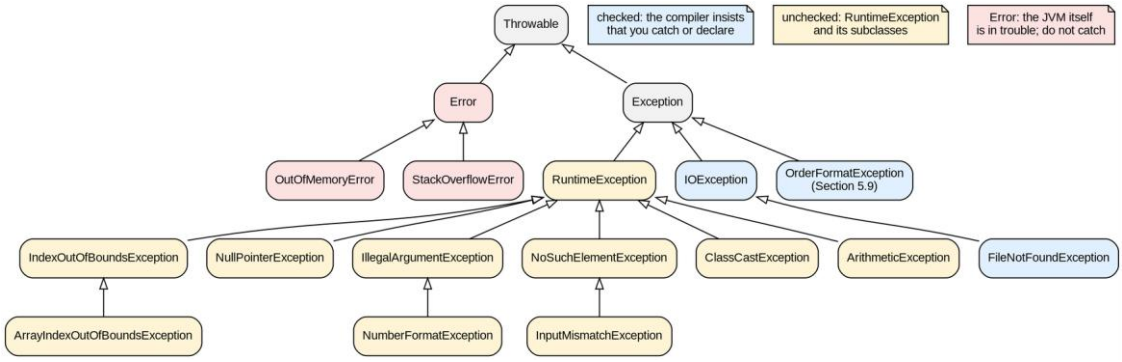


Figure 5.4 Part of Java's exception hierarchy. Blue classes are checked exceptions, yellow ones are unchecked, and red ones are errors that programs should not try to handle.

Directly below `Throwable` there are two branches.

- `Error` has a special, bad meaning in Java. If our program produces an `Error` rather than an `Exception`, something has gone so wrong that Java cannot recover: the machine has run out of memory (`OutOfMemoryError`), or a method has called itself so many times that the call stack overflowed (`StackOverflowError`). We will not catch these.

- `Exception` is the branch for the problems a program might sensibly handle. Within it, there is one more split that matters a great deal: the subclasses of `RuntimeException` on one side (yellow), and all the others (blue). Section 5.6 is devoted to that split.

Python's tree, in **Figure 5.5**, has the same idea with different names [1]. At the root is `BaseException`. Below it sit a few exceptions that mean "the program is being asked to stop", such as `SystemExit` (raised by `sys.exit`) and `KeyboardInterrupt` (raised when the user presses Ctrl+C), and then `Exception`, the ancestor of everything ordinary programs handle: `ValueError`, `TypeError`, `IndexError` and `KeyError` (both kinds of `LookupError`), `ZeroDivisionError`, and `OSError` with its children `FileNotFoundError` and `PermissionError`.

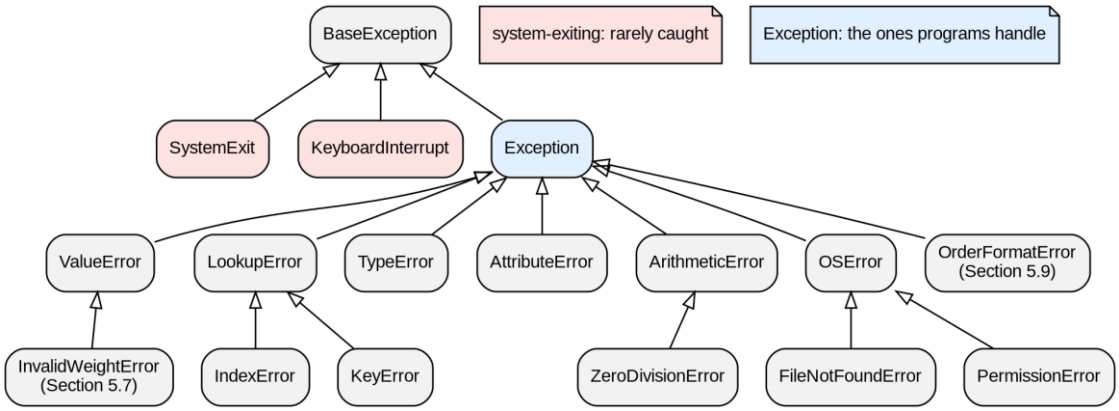


Figure 5.5 Part of Python's exception hierarchy. Programs catch subclasses of `Exception`; the exceptions beside it ask the program to stop.

Why does the family tree matter? Because a `catch` clause that names a class also catches every descendant of that class. A net labeled `IndexOutOfBoundsException` also catches `ArrayIndexOutOfBoundsException`; a net labeled `Exception` catches almost everything. We use that fact in Section 5.5.5.

5.5.4 `try` and `catch`

We have said that exceptions are thrown. To handle one, it must be *caught*. To do this, we wrap the code that could cause the exception in a `try` block and add a `catch` block that says what to do:

```
try {
    // code that could cause an exception
} catch (SomeExceptionType et) {
    // what to do when that exception occurs
}
```

Python's version uses `try` and `except`:

```
try:
    # code that could cause an exception
except SomeExceptionType as et:
    # what to do when that exception occurs
```

The rules are the same in both languages. We surround the "dangerous" code with a `try` block. If no exception occurs, the whole `try` block runs and execution continues with the code after the entire construction; the `catch` block is skipped. If an exception does occur, the rest of the `try` block is skipped and the exception's type is compared with the type in the parentheses. If it matches (meaning the exception is of that class *or of any class below it* in the family tree), the exception object is given the name we chose, `et` here, and the `catch` block runs. After the `catch` block, execution continues after the whole construction. The program has not crashed. If the type does not match, the exception carries on propagating as if the `try` were not there.

Our main aim in catching exceptions is to keep the program from quitting with an error. **Listing 5.23** repairs `InputTest`: every time the user enters something that is not a number, Java throws an exception, the `catch` block reports it, and the loop carries on, all without crashing.

Listing 5.23 `SafeInput.java`: catching `InputMismatchException` so that bad input is reported and skipped.

```
import java.util.ArrayList;
import java.util.InputMismatchException;
import java.util.Scanner;

public class SafeInput {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        ArrayList<Integer> intList = new ArrayList<>();
        System.out.println("Enter some numbers!");
        int currentVal;
        while (s.hasNext()) {                          // stop cleanly at end of input
            try {
                currentVal = s.nextInt();
                if (currentVal == 99) {
                    break;
                }
                System.out.println("Adding " + currentVal);
                intList.add(currentVal);
            } catch (InputMismatchException ime) {
                String bad = s.next();                 // discard the offending token
                System.out.println(ime + ": \"" + bad + "\" skipped, please enter only
numbers.");
            }
        }
        System.out.println("Exiting: List is " + intList);
    }
}
```

With this input, which contains two bad tokens,

```
1
2
b
3
ten
99
```

the output is:

```
Enter some numbers!
Adding 1
Adding 2
java.util.InputMismatchException: "b" skipped, please enter only numbers.
Adding 3
java.util.InputMismatchException: "ten" skipped, please enter only numbers.
Exiting: List is [1, 2, 3]
```

Three details of the listing are worth studying.

First, the most important line in the `catch` block is `String bad = s.next();`. When `nextInt()` fails, it does *not* consume the bad token; the `b` is still sitting at the front of the input. If we simply printed a message and looped, the next `nextInt()` would meet the same `b`, fail again, and the program would print the same message forever. Calling `s.next()` reads the offending token as a plain string and discards it, so the loop can move on. A popular first attempt is to create a brand-new `Scanner` in the `catch` block instead. That can appear to work when you type at the keyboard, but it is fragile: the old `Scanner` may already have read ahead and buffered input that the new one

will never see, and with input redirected from a file the new `Scanner` often finds nothing at all. Exercise 5.12 asks you to explain why. Discarding the token is the reliable repair.

Second, `ime` is the exception object, and printing it with `+` calls its `toString()` method, which gives the class name. An exception object also has `getMessage()`, which returns just the message (for `InputMismatchException` from `nextInt` there is no message at all, `getMessage()` returns `null`, which is why we print the token ourselves).

Third, the loop is `while (s.hasNext())` rather than `while (true)`. `hasNext()` returns `false` when the input is exhausted, so the program also ends gracefully if the input stops before a `99`; without it, `nextInt()` at the end of input would throw `NoSuchElementException`, which our `catch` does not name.

Listing 5.24 is the Python version. `for line in sys.stdin` reads standard input one line at a time and stops cleanly at its end, which solves the third point without any extra test.

Listing 5.24 `safe_input.py`: catching `ValueError` from `int()` in Python.

```
import sys

int_list = []
print("Enter some numbers!")
for line in sys.stdin:                          # stop cleanly at end of input
    text = line.strip()
    try:
        current_val = int(text)
        if current_val == 99:
            break
        print("Adding " + str(current_val))
        int_list.append(current_val)
    except ValueError as ve:
        print(type(ve).__name__ + ": " + repr(text) + " skipped, please enter only
numbers.")
print("Exiting: List is " + str(int_list))
```

```
Enter some numbers!
Adding 1
Adding 2
ValueError: 'b' skipped, please enter only numbers.
Adding 3
ValueError: 'ten' skipped, please enter only numbers.
Exiting: List is [1, 2, 3]
```

Because Python reads whole lines and converts them afterwards, the bad text is already consumed when `int()` fails, and nothing needs to be discarded. `type(ve).__name__` is the name of the exception's class, `ValueError`, and `repr(text)` shows the text with quotation marks.

5.5.5 Several `catch` clauses: progressively bigger nets

A single `try` may be followed by several `catch` clauses, one per kind of problem. When an exception is thrown, Java tries the clauses in order, from the top, and runs the *first* one whose type matches; the rest are skipped.

```
try {
    // code that could cause an exception
} catch (SomeExceptionType et) {
    // what to do when that exception occurs
} catch (AnotherExceptionType et2) {
    // do something else
} catch (Exception e) {
    // this will catch all other exceptions
}
```

We should think of multiple `catch` clauses as a series of progressively bigger nets, each catching a wider range of exceptions, with the first being the most specific. **Listing 5.25** tries to divide 100 by numbers taken from an array of strings, at four positions, and meets three different problems along the way.

Listing 5.25 `CatchOrder.java`: three `catch` clauses, from the most specific to the most general.

```
public class CatchOrder {
    public static void main(String[] args) {
        String[] tokens = {"12", "x7", "0"};
        int[] positions = {0, 1, 2, 5};
        for (int pos : positions) {
            try {
                int n = Integer.parseInt(tokens[pos]);
                System.out.println("position " + pos + ": 100 / " + n + " = " + (100 / n));
            } catch (NumberFormatException nfe) {          // the most specific net first
                System.out.println("position " + pos + ": not a number ("
                        + nfe.getMessage() + ")");
            } catch (ArrayIndexOutOfBoundsException aioobe) {
                System.out.println("position " + pos + ": no such position ("
                        + aioobe.getMessage() + ")");
            } catch (Exception e) {                        // the widest net last
                System.out.println("position " + pos + ": something else went wrong: " +
e);
            }
        }
        System.out.println("All positions processed.");
    }
}
```

```
position 0: 100 / 12 = 8
position 1: not a number (For input string: "x7")
position 2: something else went wrong: java.lang.ArithmeticException: / by zero
position 5: no such position (Index 5 out of bounds for length 3)
All positions processed.
```

Position 0 succeeds. At position 1, `Integer.parseInt("x7")` throws `NumberFormatException`, and the first net catches it. At position 2, the text `"0"` parses fine, but `100 / 0` throws `ArithmeticException`; neither of the first two nets fits, so the widest net, `Exception`, catches it. At position 5 the array has only three elements, and the second net catches the `ArrayIndexOutOfBoundsException` before `parseInt` is even called. After each problem the loop simply continues with the next position.

The order of the nets is not a matter of taste. If we put `catch (Exception e)` first, it would catch everything, and the more specific clauses below it could never run. Java treats that as an error and refuses to compile it, with the message “exception NumberFormatException has already been caught”; Exercise 5.9 shows it.

When two kinds of exception need exactly the same handling, Java lets one clause name both, separated by a vertical bar: `catch (NumberFormatException | ArithmeticException e)`. This is called a *multi-catch*.

Listing 5.26 is the Python version. Python checks its `except` clauses in order in the same way, and it catches several types in one clause with a tuple: `except (ValueError, ZeroDivisionError) as e:`.

Listing 5.26 `catch_order.py`: three `except` clauses, from the most specific to the most general.

```
tokens = ["12", "x7", "0"]
positions = [0, 1, 2, 5]
for pos in positions:
    try:
        n = int(tokens[pos])
        print("position " + str(pos) + ": 100 / " + str(n) + " = " + str(100 // n))
    except ValueError as ve:                # the most specific net first
        print("position " + str(pos) + ": not a number (" + str(ve) + ")")
    except IndexError as ie:
        print("position " + str(pos) + ": no such position (" + str(ie) + ")")
    except Exception as e:                  # the widest net last
        print("position " + str(pos) + ": something else went wrong: " + repr(e))
print("All positions processed.")
```

```
position 0: 100 / 12 = 8
position 1: not a number (invalid literal for int() with base 10: 'x7')
position 2: something else went wrong: ZeroDivisionError('integer division or modulo by
zero')
position 5: no such position (list index out of range)
All positions processed.
```

> **PITFALL.** A net labeled `Exception` catches problems you never thought of, including bugs in your own code, and so it can hide them. Use it only as the last clause, and only where continuing really is the right thing to do, as in a loop that processes many independent items. Never write an *empty* `catch` block: it throws away the one piece of information that would have told you what went wrong.

5.5.6 `finally`: the block that always runs

Sometimes there is work that must be done whether or not an exception occurred: closing a file, releasing a network connection, turning off a machine's heater. For this, a `try` statement may end with a `finally` block. The code in `finally` runs in every case: after the `try` block completes normally, after a `catch` block has handled an exception, and even when an exception is *not* caught and is about to propagate onward, or when the `try` block executes `return`. **Figure 5.6** traces all the paths.

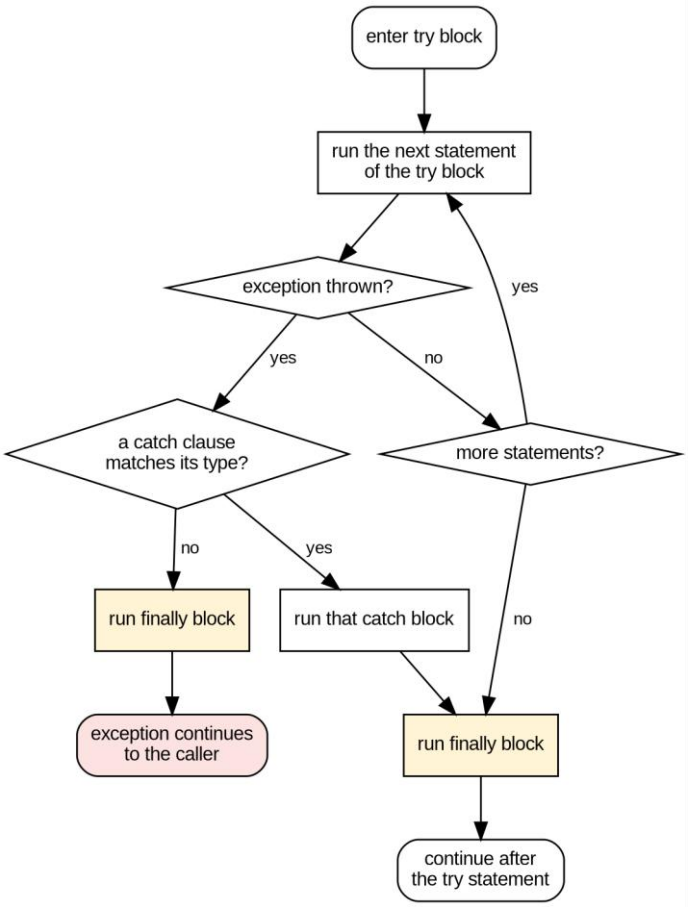


Figure 5.6 The paths through a `try`–`catch`–`finally` statement. Every path passes through a `finally` block.

Listing 5.27 demonstrates two of these paths with a `divide` method that has a `try` and a `finally` but no `catch`.

Listing 5.27 `FinallyDemo.java`: `finally` runs after a normal `return` and when an exception escapes.

```
public class FinallyDemo {
    static int divide(int a, int b) {
        try {
            System.out.println("  try: dividing " + a + " by " + b);
            return a / b;
        } finally {
            System.out.println("  finally: this line always runs");
        }
    }

    public static void main(String[] args) {
        System.out.println("Case 1: no exception");
        System.out.println("  result = " + divide(10, 2));

        System.out.println("Case 2: an exception, caught by the caller");
        try {
            System.out.println("  result = " + divide(1, 0));
        } catch (ArithmeticException e) {
            System.out.println("  caught in main: " + e.getMessage());
        }
        System.out.println("Done.");
    }
}
```

```
Case 1: no exception
  try: dividing 10 by 2
  finally: this line always runs
  result = 5
Case 2: an exception, caught by the caller
  try: dividing 1 by 0
  finally: this line always runs
  caught in main: / by zero
Done.
```

In case 1 the `try` block reaches `return a / b;`. The value 5 is computed and is about to be returned, but before the method actually hands it back, the `finally` block runs and prints its line; only then does `main` receive 5 and print `result = 5`. In case 2, `1 / 0` throws `ArithmeticException`. There is no `catch` in `divide`, so the exception will propagate to `main`, but again, the `finally` block runs first, on the way out. Then `main`'s `catch` handles the exception. Note that `result =` is never printed in case 2: the `println` in `main` was waiting for a value that never came.

Python's `try` has the same `finally`, and one more optional clause, `else`, which runs only when the `try` block raised nothing. **Listing 5.28** adds a third case to show it.

Listing 5.28 `finally_demo.py`: `finally` in Python, and the `else` clause.

```
def divide(a, b):
    try:
        print("  try: dividing " + str(a) + " by " + str(b))
        return a // b
    finally:
        print("  finally: this line always runs")


def main():
    print("Case 1: no exception")
    print("  result = " + str(divide(10, 2)))

    print("Case 2: an exception, caught by the caller")
    try:
        print("  result = " + str(divide(1, 0)))
    except ZeroDivisionError as e:
        print("  caught in main: " + str(e))

    print("Case 3: Python's else clause")
    try:
        result = divide(9, 3)
    except ZeroDivisionError:
        print("  never printed here")
    else:
        print("  else: runs only when the try block raised nothing, result = " +
str(result))
    print("Done.")


if __name__ == "__main__":
    main()
```

```
Case 1: no exception
  try: dividing 10 by 2
  finally: this line always runs
  result = 5
Case 2: an exception, caught by the caller
  try: dividing 1 by 0
  finally: this line always runs
  caught in main: integer division or modulo by zero
Case 3: Python's else clause
  try: dividing 9 by 3
  finally: this line always runs
  else: runs only when the try block raised nothing, result = 3
Done.
```

The order in the full Python form is `try`, then any `except` clauses, then `else`, then `finally`. Putting code in `else` rather than at the end of `try` is a way of saying "only this part of the code is expected to raise the exceptions I am catching", which keeps the nets from catching more than they should.

5.5.7 Throwing an exception yourself

Exceptions are not only thrown by the library. Sometimes we want to force one to happen ourselves, when a caller does something we want to forbid. For this, Java has the keyword `throw` and Python the keyword `raise`, each followed by a new exception object:

```
throw new IllegalArgumentException("The numbers you've entered are not acceptable.");
raise ValueError("The numbers you've entered are not acceptable.")
```

`IllegalArgumentException` is very common in Java wherever a parameter must lie within certain bounds; its Python counterpart is `ValueError`. The classic example is a grayscale value, the brightness of one pixel in a black-and-white image, which must lie between 0 and 255. **Listing 5.29** has a method that throws an exception for a value that exceeds 255, and a `main` that catches it and, instead of halting halfway through, resets the value to 255 and carries on.

Listing 5.29 `IllegalArgumentExceptionExample.java`: throwing `IllegalArgumentException` for an out-of-range grayscale value, and recovering by clamping it to 255.

```java
public class IllegalArgumentExceptionExample {
    public static boolean isExceeded(int x) {
        boolean result = false;
        if (x > 255) {
            throw new IllegalArgumentException("Value exceeds 255");
        } else {
            result = true;
        }
        return result;
    }

    public static void main(String[] args) {
        int value = 270;
        try {
            // Assume, we deal with Grayscale value: 0-255
            // Check the value's range
            if (isExceeded(value)) {
                System.out.println(value + " is a valid Grayscale value");
            }
        } catch (IllegalArgumentException ie) {
            System.out.println(value + " is an invalid Grayscale value");
            System.out.println("Resetting the value to 255");
            value = 255;
        }
        System.out.println("Continue processing the value...");
    }
}
```

```
270 is an invalid Grayscale value
Resetting the value to 255
Continue processing the value...
```

In `main` the value is 270. `isExceeded(270)` finds that $270 > 255$ and throws; the `if` never completes, and control jumps to the `catch` block, which prints two lines and sets the value to 255. Then execution continues below the `try` statement. Often we might just print that an error has occurred and let the user continue below the `try` block; here we did something slightly smarter and repaired the value.

The listing is kept exactly as a first version might be written, and it contains a lesson in naming. A method called `isExceeded` would be expected to return `true` when the value *is* exceeded; this one returns `true` when the value is fine and throws when it is exceeded. The program works, but the next reader will misread it. A better name would be `checkGrayscale`, returning nothing and simply throwing on bad input; Section 5.11 comes back to this.

Listing 5.30 is the Python version, which raises `ValueError`.

Listing 5.30 `grayscale.py`: raising `ValueError` for an out-of-range grayscale value.

```
def is_exceeded(x):
    result = False
    if x > 255:
        raise ValueError("Value exceeds 255")
    else:
        result = True
    return result


def main():
    value = 270
    try:
        # Assume, we deal with Grayscale value: 0-255
        # Check the value's range
        if is_exceeded(value):
            print(str(value) + " is a valid Grayscale value")
    except ValueError:
        print(str(value) + " is an invalid Grayscale value")
        print("Resetting the value to 255")
        value = 255
    print("Continue processing the value...")


if __name__ == "__main__":
    main()
```

```
270 is an invalid Grayscale value
Resetting the value to 255
Continue processing the value...
```

Two remarks of principle complete this section. First, if we throw an exception, it must be caught somewhere, or the program crashes because of it. Second, it is good practice to throw exceptions only to signal that something erroneous has happened. Exceptions should not be used as a clever way of returning from a loop or passing ordinary values around: they are slower than ordinary control flow and, worse, they confuse every reader who expects an exception to mean an error [2].

5.6 Checked and Unchecked Exceptions

The family tree in **Figure 5.4** raises an interesting point. There are two sorts of exception in Java: those that can occur almost anywhere, and so cannot really be planned for by the compiler, and those that Java *knows* can occur at a particular place, and so forces us to do something about.

5.6.1 Unchecked exceptions: usually our own mistakes

We have met several exceptions already. Remember them? **Listing 5.31** provokes three of the most famous, each inside its own `try` so that we can see all three messages.

Listing 5.31 `UncheckedDemo.java`: three unchecked exceptions caused by programming mistakes.

```
public class UncheckedDemo {
    static String[] arr = new String[6];

    public static void main(String[] args) {
        try {
            System.out.println(arr[6]);                  // What happens here?
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("1: " + e);
        }
        try {
            System.out.println(arr[3].length());         // What about here?
        } catch (NullPointerException e) {
            System.out.println("2: " + e);
        }
        Object x = Integer.valueOf(0);
        try {
            System.out.println((String) x);              // How about now?
        } catch (ClassCastException e) {
            System.out.println("3: " + e);
        }
    }
}
```

```
1: java.lang.ArrayIndexOutOfBoundsException: Index 6 out of bounds for length 6
2: java.lang.NullPointerException: Cannot invoke "String.length()" because
"UncheckedDemo.arr[3]" is null
3: java.lang.ClassCastException: class java.lang.Integer cannot be cast to class
java.lang.String (java.lang.Integer and java.lang.String are in module java.base of loader
'bootstrap')
```

The first is familiar: an array of length 6 has no index 6. The second is the notorious `NullPointerException`: `arr[3]` still holds its default value `null`, there is no string there, and so there is nothing on which to call `length()`. Modern Java explains this very helpfully: `Cannot invoke "String.length()" because "UncheckedDemo.arr[3]" is null`. The third is `ClassCastException`: the variable `x` has type `Object`, which is allowed to refer to anything, and it refers to an `Integer`; the cast `(String) x` claims that it is a `String`, and Java checks the claim at run time and finds it false.

All three are *unchecked exceptions*. They all extend `RuntimeException`, and the compiler does not force us to catch them. There are two reasons. First, the compiler cannot usually tell whether they will occur: almost any expression with an array index, a dot or a cast might throw one, and demanding a `try` around every such expression would make programs unreadable. Second, and more importantly, unchecked exceptions are generally caused by programmer error. That much should be clear from the three examples: in each one the program, not the user, did something that could never work.

> **KEY INSIGHT.** If our own code produces an unchecked exception, it is usually our fault, and the right response is to fix the code: check the index, initialize the variable, test the type. Simply catching these exceptions may not get to the root of the problem at all.

Python has the same three mistakes, with different names, as **Listing 5.32** shows. Using an index that is too large raises `IndexError`; asking for the length of `None` raises `TypeError`; asking an `int` for a method it does not have raises `AttributeError`. (Python has no casts, so the nearest equivalent to a `ClassCastException` is using a value as though it were of a type it is not.)

Listing 5.32 `unchecked_demo.py`: the same three kinds of mistake in Python.

```
arr = [None] * 6

try:
    print(arr[6])                           # What happens here?
except IndexError as e:
    print("1:", repr(e))
try:
    print(len(arr[3]))                      # What about here?
except TypeError as e:
    print("2:", repr(e))
x = 0
try:
    print(x.upper())                        # How about now?
except AttributeError as e:
    print("3:", repr(e))
```

```
1: IndexError('list index out of range')
2: TypeError("object of type 'NoneType' has no len()")
3: AttributeError("'int' object has no attribute 'upper'")
```

5.6.2 Checked exceptions: the compiler insists

Checked exceptions are different. Pretty much everything in the package `java.io` can throw them, and they represent problems that are nobody's programming error: we should be able to see the obvious potential for exceptions when trying to read from a file. What if the file isn't there? Java obviously cannot read from it, and so it has to raise an exception. What if the disk is full, or the file belongs to another user, or the network drive has just disconnected? No amount of careful programming can prevent these; a correct program can only *plan* for them.

For such exceptions Java makes the planning compulsory. They extend `Exception` but not `RuntimeException` (the blue boxes of **Figure 5.4**), and they are called *checked exceptions* because the compiler checks that each one is dealt with. If we try to compile the following,

```
import java.io.*;

public class Test {
    public static void main(String[] args) {
        BufferedReader br = new BufferedReader(new FileReader("filename.txt"));
    }
}
```

compilation fails. The compiler tells us:

```
Test.java:5: error: unreported exception FileNotFoundException; must be caught or declared
to be thrown
       BufferedReader br = new BufferedReader(new FileReader("filename.txt"));
                                              ^
1 error
```

The wording of that sentence deserves careful reading. The constructor `FileReader(String)` is declared as one that may throw `FileNotFoundException`. Our code must therefore do one of two things:

1. **catch it**: surround the call with `try` and a `catch` that handles `FileNotFoundException` (or one of its ancestors); or
2. **declare it**: say, in the method's header, that this method deliberately does not handle the exception but passes it on to whoever called the method. The declaration uses the keyword `throws`: `static String firstLine(String fileName) throws IOException`.

Letting `main` itself declare `throws FileNotFoundException` would compile, but it would be a very poor idea: `main` has no caller in our program, so the exception would simply crash the program

with a stack trace, which is exactly what we are trying to avoid. Instead, we should handle the possibility that the file is not there. **Listing 5.33** does both things, each in the right place.

Listing 5.33 `OpenFile.java`: `firstLine` declares `IOException` with `throws`; `main` catches it.

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class OpenFile {
    // Declares the checked exception instead of handling it: the caller decides.
    static String firstLine(String fileName) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(fileName));
        String line = br.readLine();
        br.close();
        return line;
    }

    public static void main(String[] args) {
        String[] names = {"filename.txt", "notes.txt"};
        for (String name : names) {
            try {
                System.out.println(name + " begins: " + firstLine(name));
            } catch (FileNotFoundException fnfe) {
                System.out.println("File not found: " + fnfe.getMessage());
            } catch (IOException ioe) {
                System.out.println("Miraculously, some other IO exception: "
                        + ioe.getMessage());
            }
        }
    }
}
```

```
File not found: filename.txt (No such file or directory)
notes.txt begins: Profitina Laundry opens at 07:00.
```

`firstLine` is a small helper that opens a file, reads its first line and closes it. It is not the right place to decide what to do about a missing file: in one program that means “print a friendly message”, in another “use default settings”, in a third “ask the user for another name”. So `firstLine` *declares* `throws IOException` and leaves the decision to its caller. `main` knows what it wants, a message, and so `main` *catches*. The first file does not exist, and we see the message `File not found: filename.txt (No such file or directory)`, where the part in parentheses comes from the operating system. The second file, `notes.txt`, exists in the same folder:

```
Profitina Laundry opens at 07:00.
Express orders are ready in six hours.
The courier leaves at 16:00.
```

and its first line is printed.

Notice that we could not simply write `catch (FileNotFoundException fnfe)` alone: `readLine()` and `close()` may throw the more general `IOException`, and the compiler insists that it too be handled. Hence the second clause, whose message (“Miraculously, some other IO exception”) admits that we do not expect it. `FileNotFoundException` extends `IOException`, so the specific net must come first.

There is also a lesson in what this code does *not* do well. If `readLine()` threw an exception, `br.close()` would be skipped and the file would stay open. Section 5.8.2 shows how to guarantee that a file is closed.

A related, very common mistake is to write the `try` block around the opening of the file only, and then try to read from the reader *after* the `try`:

```
try {
    BufferedReader br = new BufferedReader(new FileReader("filename.txt"));
} catch (FileNotFoundException fnfe) {
    System.out.println("File not found: " + fnfe.getMessage());
}
// read from br here?  No: br does not exist outside the try block.
```

This compiles, but it is useless, for two reasons. The variable `br` is declared inside the `try` block and does not exist after the closing brace; and even if it did, when the file is missing the program would print its message and then go on to use a reader that was never created. The reading belongs *inside* the `try`, or in a method that declares the exception, exactly as **Listing 5.33** does.

5.6.3 Catch, or declare?

With a checked exception you always have the two choices of Section 5.6.2, and choosing well is a skill. A good rule of thumb:

- **Catch** the exception in the method that knows *what to do* about it: show a message, try another file, use a default, skip the bad record.
- **Declare** it in methods that do not know. A method that opens and reads a file on someone else's behalf usually cannot decide what a missing file means, so it declares `throws IOException` and lets the exception propagate to a caller who can.

The compiler's insistence is not bureaucracy. When you call a method whose header says `throws IOException`, the compiler makes sure you cannot forget that the call can fail. That is precisely why the Java designers made I/O exceptions checked: failures of the outside world are normal, and a program that ignores them is wrong [3].

5.6.4 Python's position

Python has no checked exceptions. Every exception behaves like Java's unchecked ones: the interpreter never forces a caller to catch anything, and there is no `throws` clause. **Listing 5.34** is the Python version of **Listing 5.33**.

Listing 5.34 `open_file.py`: `first_line` documents the exception in its docstring; `main` catches it.

```
def first_line(file_name):
    """Return the first line of a text file.

    Raises:
        OSError: if the file cannot be opened or read (FileNotFoundError is one kind).
    """
    f = open(file_name, encoding="utf-8")
    line = f.readline().rstrip("\n")
    f.close()
    return line


def main():
    names = ["filename.txt", "notes.txt"]
    for name in names:
        try:
            print(name + " begins: " + first_line(name))
        except FileNotFoundError as fnfe:
            print("File not found: " + str(fnfe))
        except OSError as ose:
            print("Miraculously, some other OS error: " + str(ose))


if __name__ == "__main__":
    main()
```

```
File not found: [Errno 2] No such file or directory: 'filename.txt'
notes.txt begins: Profitina Laundry opens at 07:00.
```

Since the language will not remind the caller, Python programmers state in the documentation which exceptions a function may raise; the conventional place is a `Raises:` section in the function's docstring, as in `first_line`. Chapter 6 treats docstrings properly. The discipline is the same as in Java; only the enforcement is missing, so it is up to you.

Which approach is better is a debate as old as Java itself. Checked exceptions make failures visible and hard to ignore; critics say they clutter code and tempt programmers to swallow exceptions with empty `catch` blocks just to silence the compiler. Both camps agree on the essentials, which is what this chapter teaches: know which operations can fail, handle each failure in the place that knows what to do, and never hide one [2].

5.7 Exceptions of Your Own

The library's exceptions describe the library's problems. A laundry program has problems of its own: a weight that is not a number, a weight of −1 kilograms, an order for a service the laundry does not offer. We could report all of them with `IllegalArgumentException` or `ValueError`, but a *custom exception class* has three advantages. Its name says exactly what went wrong; it can carry extra information, such as the bad text or the line number in a file; and a caller can catch it specifically, without also catching unrelated problems.

Writing one is short. In Java we write a class that extends `Exception`, for a checked exception, or `RuntimeException`, for an unchecked one, and pass the message to the parent class with `super(message)`. We choose *checked* when the failure is an expected outcome that callers must plan for, such as bad data from outside the program, and *unchecked* when it signals a programming mistake. An invalid weight typed by a user is the former. **Listing 5.35** defines `InvalidWeightException` and uses it.

Listing 5.35 `WeightCheck.java`: a custom checked exception that carries the offending text and, where there is one, its cause.

```
public class WeightCheck {
    // A checked exception of our own, nested here so that the file stays whole.
    static class InvalidWeightException extends Exception {
        private final String weightText;

        InvalidWeightException(String weightText, String message, Throwable cause) {
            super(message, cause);
            this.weightText = weightText;
        }

        String getWeightText() {
            return weightText;
        }
    }

    static double parseWeight(String text) throws InvalidWeightException {
        double kg;
        try {
            kg = Double.parseDouble(text);
        } catch (NumberFormatException nfe) {
            throw new InvalidWeightException(text, "not a number: " + text, nfe);
        }
        if (kg <= 0 || kg > 50) {
            throw new InvalidWeightException(text, "out of range 0-50 kg: " + text, null);
        }
        return kg;
    }

    public static void main(String[] args) {
        String[] inputs = {"3.5", "abc", "-1", "80", "12"};
        for (String input : inputs) {
            try {
                System.out.println("OK       " + parseWeight(input) + " kg");
            } catch (InvalidWeightException e) {
                System.out.println("REJECTED \"" + e.getWeightText() + "\": " +
e.getMessage());
                if (e.getCause() != null) {
                    System.out.println("         caused by " + e.getCause());
                }
            }
        }
    }
}
```

```
OK       3.5 kg
REJECTED "abc": not a number: abc
         caused by java.lang.NumberFormatException: For input string: "abc"
REJECTED "-1": out of range 0-50 kg: -1
REJECTED "80": out of range 0-50 kg: 80
OK       12.0 kg
```

Some points to notice:

- To keep the program in one file, `InvalidWeightException` is written as a *nested class*, a class declared inside `WeightCheck` with the modifier `static`. It behaves like any other class. In a larger program it would normally have a file of its own, as `OrderFormatException` will in Section 5.9.
- The constructor calls `super(message, cause)`. The class `Exception` already knows how to store a message and a *cause*, so we reuse that instead of writing our own. We add one field of our own, `weightText`, with a getter.
- `parseWeight` declares `throws InvalidWeightException`. It is a checked exception, so the compiler now makes sure every caller of `parseWeight` handles it, which `main` does.
- When `Double.parseDouble("abc")` fails, we catch its `NumberFormatException` and throw our own exception in its place, passing the original as the *cause*. This is called *exception chaining* or

wrapping. The caller sees one clear, laundry-level message, "not a number: abc", while nothing is lost: `getCause()` still returns the original exception, and a stack trace would show both. Translating a low-level exception into one that makes sense at the caller's level is a good habit [2].

- For −1 and 80 there is no underlying exception: the number is fine as a number but wrong as a laundry weight. We pass `null` as the cause.

In Python a custom exception is a class that extends `Exception` or one of its subclasses. Because an invalid weight really is a kind of bad value, **Listing 5.36** extends `ValueError`, so that code that already catches `ValueError` will catch it too.

Listing 5.36 `weight_check.py`: a custom exception class in Python, chained with `raise ... from`.

```
class InvalidWeightError(ValueError):
    """Raised when a laundry weight is not a number or is out of range."""

    def __init__(self, weight_text, message):
        super().__init__(message)
        self.weight_text = weight_text


def parse_weight(text):
    try:
        kg = float(text)
    except ValueError as ve:
        raise InvalidWeightError(text, "not a number: " + text) from ve
    if kg <= 0 or kg > 50:
        raise InvalidWeightError(text, "out of range 0-50 kg: " + text)
    return kg


def main():
    inputs = ["3.5", "abc", "-1", "80", "12"]
    for text in inputs:
        try:
            print("OK       " + str(parse_weight(text)) + " kg")
        except InvalidWeightError as e:
            print('REJECTED "' + e.weight_text + '": ' + str(e))
            if e.__cause__ is not None:
                print("         caused by " + repr(e.__cause__))


if __name__ == "__main__":
    main()
```

```
OK       3.5 kg
REJECTED "abc": not a number: abc
         caused by ValueError("could not convert string to float: 'abc'")
REJECTED "-1": out of range 0-50 kg: -1
REJECTED "80": out of range 0-50 kg: 80
OK       12.0 kg
```

`super().__init__(message)` hands the message to the parent class, so that `str(e)` returns it. The chaining uses the syntax `raise NewError(...) from original`; afterwards, the original exception is available as `e.__cause__`, and an uncaught traceback would print both, joined by the sentence "The above exception was the direct cause of the following exception".

> **KEY INSIGHT.** A good exception answers three questions for whoever catches it: *what kind* of problem (its class), *what exactly* (its message), and *with which data* (its extra fields and its cause). Design your own exceptions to answer all three.

5.8 Files: Streams, Readers, and Writers

5.8.1 Streams and readers

Most input and output in Java is done with *streams*. A stream is a sequence of data that flows from a source, such as the keyboard, a file or a network connection, into the program (an *input stream*), or from the program to a destination (an *output stream*). `System.in`, which we have been handing to `Scanner`, is an `InputStream`: a stream of raw bytes from the keyboard.

Bytes are not yet text. An `InputStreamReader` wraps an input stream and turns its bytes into characters:

```
InputStreamReader isr = new InputStreamReader(System.in);
char character = (char) isr.read();   // read a single character
```

Reading one character at a time is tiresome, so we wrap once more. A `BufferedReader` wraps any reader, collects characters in a *buffer*, a block of memory, and hands them out a whole line at a time with `readLine()`:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

This wrapping of one object inside another, each adding one ability, is a pattern you will see throughout Java's I/O library. **Listing 5.37** uses it to read lines into an `ArrayList` until the user types `stop`.

Listing 5.37 `ReadUntilStop.java`: reading lines from the keyboard with a `BufferedReader` until `stop` or the end of input.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class ReadUntilStop {
    public static void main(String[] args) {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        ArrayList<String> stringList = new ArrayList<>();
        try {
            String currentLine = br.readLine();
            while (currentLine != null && !currentLine.equals("stop")) {
                stringList.add(currentLine);
                currentLine = br.readLine();
            }
        } catch (IOException e) {
            System.out.println("Could not read the keyboard: " + e.getMessage());
        }
        // Print the list out
        System.out.println("Read " + stringList.size() + " line(s): " + stringList);
    }
}
```

With the input file

```
wash 3 kg
iron 2 shirts
express please
stop
this line is never read
```

the output is:

```
Read 3 line(s): [wash 3 kg, iron 2 shirts, express please]
```

Two differences from `Scanner` stand out. First, `readLine()` returns `null` at the end of the input instead of throwing an exception, so the loop tests `currentLine != null` before it looks at the line. (Testing in that order matters: if the line is `null`, the `&&` stops there and never calls `equals` on `null`, which would throw `NullPointerException`.) Second, `readLine()` can throw the checked `IOException`, so the compiler insists on the `try` and `catch`.

A compact form of the same loop, which you will meet in other people's code, reads and tests in a single condition:

```
while (!(currentLine = br.readLine()).equals("stop")) {
    stringList.add(currentLine);
}
```

It works, but the version in the listing is easier to understand and also handles the end of input.

In Python, `sys.stdin` is already a text stream with its own buffer, so there is nothing to wrap. **Listing 5.38** reads a line with `sys.stdin.readline()`, which returns the empty string `""` at the end of input. (An empty *line* typed by the user is not `""` but `"\n"`, because the line break is included, which is why the two can be told apart.)

Listing 5.38 `read_until_stop.py`: reading lines from standard input until `stop` or the end of input.

```
import sys

string_list = []
current_line = sys.stdin.readline()          # "" means end of input
while current_line != "" and current_line.rstrip("\n") != "stop":
    string_list.append(current_line.rstrip("\n"))
    current_line = sys.stdin.readline()
# Print the list out
print("Read " + str(len(string_list)) + " line(s): " + str(string_list))
```

```
Read 3 line(s): ['wash 3 kg', 'iron 2 shirts', 'express please']
```

One particularly useful property of readers is that they can be applied to anything that provides an input stream. The same `BufferedReader` can read from a web page, by wrapping the stream that a `java.net.URL` or the newer `HttpClient` opens, or from a file, by wrapping a `FileReader`:

```
BufferedReader fr  = new BufferedReader(new FileReader(new
File("/path/to/my/filename.txt")));
// equivalent to
BufferedReader fr2 = new BufferedReader(new FileReader("/path/to/my/filename.txt"));
```

Which should we use, `Scanner` or `BufferedReader`? They overlap, and both are fine. `Scanner` is the more convenient for small interactive programs, because it can read numbers directly (`nextInt`, `nextDouble`) and split input into tokens. `BufferedReader` is simpler and considerably faster for reading large amounts of text line by line, it reports the end of input with `null` rather than an exception, and it forces you to face `IOException` honestly. A common and sensible practice is `Scanner` for keyboard dialogues and `BufferedReader` for files, which is what the rest of this chapter does. In Python there is no such choice to make: `input()` for dialogues, and iterating over a file object for files.

5.8.2 Closing what you open

Every open file uses a resource of the operating system, a *file handle*, and the number of handles a program may hold is limited. Output streams also keep data in a buffer and only guarantee that it reaches the disk when the stream is *closed*. Therefore every stream must be closed once you are

done with it, whether or not an exception happened along the way. That is exactly the kind of work that belongs in a `finally` block.

Listing 5.39 copies a file one byte at a time, using `FileInputStream` and `FileOutputStream` (streams of raw bytes, suitable for any kind of file), and closes both in `finally`. This is the way it had to be done before Java 7, and you will still see it in older code.

Listing 5.39 `CopyFileClassic.java`: copying a file and closing both streams in `finally`, the pre-Java-7 way.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyFileClassic {
    public static void main(String[] args) {
        String from = args.length > 0 ? args[0] : "notes.txt";
        String to = args.length > 1 ? args[1] : "notes_copy.txt";
        int exitCode = 0;
        int count = 0;
        FileInputStream source = null;
        FileOutputStream target = null;
        try {
            source = new FileInputStream(from);
            target = new FileOutputStream(to);
            for (int c = source.read(); c != -1; c = source.read()) {
                target.write(c);                    // one byte at a time
                count++;
            }
        } catch (IOException e) {
            System.err.println("IO error " + from);
            System.err.println(e.getMessage());
            exitCode = 1;                           // we've changed our mind
        } finally {
            try {
                if (source != null) {
                    source.close();
                }
            } catch (IOException e) { /* nothing sensible left to do */ }
            try {
                if (target != null) {
                    target.close();
                }
            } catch (IOException e) { /* nothing sensible left to do */ }
        }
        System.out.println("Copied " + count + " bytes from " + from + " to " + to
                + ", exit code " + exitCode);
        System.exit(exitCode);
    }
}
```

```
Copied 102 bytes from notes.txt to notes_copy.txt, exit code 0
```

`notes.txt` has 102 bytes (99 characters plus three line breaks, as the solution to Exercise 5.13 confirms), and all 102 were copied. Read the structure carefully, because it is harder than it looks.

- Both variables are declared *before* the `try`, and set to `null`, so that the `finally` block can see them. A variable declared inside the `try` block would not exist in `finally`.
- In `finally` we must check for `null`, because an exception may have happened before a stream was opened. If `notes.txt` did not exist, `source` would still be `null` and `target` would never have been created; calling `close()` on `null` would throw `NullPointerException` from inside `finally`, hiding the real problem.

- `close()` itself can throw `IOException`, so each call needs its own `try` and `catch`, *inside* the `finally`. They are separate so that a failure closing `source` does not prevent us from closing `target`.
- The error messages go to `System.err`, the standard *error* stream, which is normally also the screen but can be redirected separately from the ordinary output.

Nearly half the code is concerned with closing. This could get complicated, and in practice programmers often got it subtly wrong. **Listing 5.40** is the same pattern in Python, where `close()` on a file does not raise in normal use, so the `finally` block is shorter.

Listing 5.40 `copy_file_classic.py`: the same copy with explicit `close()` calls in `finally`.

```
import sys


def main():
    source_name = sys.argv[1] if len(sys.argv) > 1 else "notes.txt"
    target_name = sys.argv[2] if len(sys.argv) > 2 else "notes_copy.txt"
    exit_code = 0
    count = 0
    source = None
    target = None
    try:
        source = open(source_name, "rb")
        target = open(target_name, "wb")
        c = source.read(1)
        while c != b"":                     # one byte at a time
            target.write(c)
            count += 1
            c = source.read(1)
    except OSError as e:
        print("IO error " + source_name, file=sys.stderr)
        print(e, file=sys.stderr)
        exit_code = 1                       # we've changed our mind
    finally:
        if source is not None:
            source.close()
        if target is not None:
            target.close()
    print("Copied " + str(count) + " bytes from " + source_name + " to " + target_name
          + ", exit code " + str(exit_code))
    sys.exit(exit_code)


if __name__ == "__main__":
    main()
```

```
Copied 102 bytes from notes.txt to notes_copy.txt, exit code 0
```

5.8.3 `try`-with-resources and `with`

Because the pattern of **Listing 5.39** is so common and so easy to get wrong, Java 7 added a better one: *`try`-with-resources*. Any object whose class implements the interface `AutoCloseable` (all streams, readers and writers do) can be opened in parentheses right after the word `try`. Java then guarantees to close it when the `try` block ends, whether it ends normally, by `return`, or by an exception, and it closes several resources in the reverse order of opening. **Listing 5.41** is the modern version of the copy. It also moves up to 8 KB per `read` call instead of one byte, which is enormously faster for large files.

Listing 5.41 `CopyFile.java`: copying with `try`-with-resources, which closes both streams automatically.

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyFile {
    static long copy(String from, String to) throws IOException {
        long count = 0;
        byte[] buffer = new byte[8192];             // move up to 8 KB per call
        try (FileInputStream source = new FileInputStream(from);
             FileOutputStream target = new FileOutputStream(to)) {
            int n;
            while ((n = source.read(buffer)) != -1) {
                target.write(buffer, 0, n);
                count += n;
            }
        }                                           // both streams closed here, always
        return count;
    }

    public static void main(String[] args) {
        String[][] jobs = {{"notes.txt", "notes_copy2.txt"}, {"nothing.txt", "x.txt"}};
        for (String[] job : jobs) {
            try {
                long n = copy(job[0], job[1]);
                System.out.println("Copied " + n + " bytes from " + job[0] + " to " +
job[1]);
            } catch (IOException e) {
                System.out.println("Could not copy " + job[0] + ": " + e.getMessage());
            }
        }
    }
}
```

```
Copied 102 bytes from notes.txt to notes_copy2.txt
Could not copy nothing.txt: nothing.txt (No such file or directory)
```

All the closing code has disappeared, and yet the guarantee is stronger than before. Notice too the division of labor from Section 5.6.3: `copy` declares `throws IOException`, because it does not know what a failure should mean, and `main` catches it and reports it. The second job asks for a file that does not exist, and the program reports the fact and carries on.

`source.read(buffer)` fills as much of the array `buffer` as it can and returns how many bytes it placed there, or -1 at the end of the file. `target.write(buffer, 0, n)` writes exactly those `n` bytes, not the whole buffer, because the last read usually fills it only partly.

Python has had the same idea since version 2.5: the `with` statement. An object used after `with` is a *context manager*, and a file object is one: when the `with` block ends, for whatever reason, the file is closed. **Listing 5.42** opens both files in a single `with`.

Listing 5.42 `copy_file.py`: copying with the `with` statement.

```
def copy(source_name, target_name):
    count = 0
    with open(source_name, "rb") as source, open(target_name, "wb") as target:
        while True:
            chunk = source.read(8192)       # move up to 8 KB per call
            if not chunk:
                break
            target.write(chunk)
            count += len(chunk)
    return count                            # both files closed here, always


def main():
    jobs = [("notes.txt", "notes_copy2.txt"), ("nothing.txt", "x.txt")]
    for source_name, target_name in jobs:
        try:
            n = copy(source_name, target_name)
            print("Copied " + str(n) + " bytes from " + source_name + " to " + target_name)
        except OSError as e:
            print("Could not copy " + source_name + ": " + str(e))


if __name__ == "__main__":
    main()
```

```
Copied 102 bytes from notes.txt to notes_copy2.txt
Could not copy nothing.txt: [Errno 2] No such file or directory: 'nothing.txt'
```

> **KEY INSIGHT.** In new code, open every file with `try`-with-resources in Java and with `with` in Python. Write a `finally` block by hand only for resources that these mechanisms cannot manage.

5.8.4 Writing and appending text

For writing *text* we use writers, the character counterparts of output streams. `FileWriter` writes characters to a file; wrapping it in a `PrintWriter` gives us the familiar `print`, `println` and `printf`. A `FileWriter` normally *replaces* the file's contents (or creates the file if it does not exist); giving `true` as an extra argument opens it in *append* mode instead, so that new text goes after what is already there. Python expresses the same choices with the *mode* argument of `open`: `"r"` to read (the default), `"w"` to write from scratch, `"a"` to append.

Profitina Laundry prints a receipt. **Listing 5.43** builds the text of the receipt in memory, writes it to `receipt.txt`, appends a thank-you line, and then reads the file back, numbering the lines.

Listing 5.43 `ReceiptWriter.java`: building text with `StringBuilder`, writing a file, appending to it, and reading it back.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.nio.charset.StandardCharsets;

public class ReceiptWriter {
    public static void main(String[] args) {
        String[] items = {"WASH 3.5 kg", "IRON 2.0 kg", "EXPRESS surcharge"};
        long[] prices = {24500, 10000, 17250};

        StringBuilder sb = new StringBuilder();       // build the text once, cheaply
        long total = 0;
        for (int i = 0; i < items.length; i++) {
            sb.append(String.format("%-20s %,10d%n", items[i], prices[i]));
            total += prices[i];
        }
        sb.append(String.format("%-20s %,10d%n", "TOTAL (Rp)", total));

        try (PrintWriter out = new PrintWriter(
                new FileWriter("receipt.txt", StandardCharsets.UTF_8))) {
            out.print(sb);                            // "w": create or overwrite
        } catch (IOException e) {
            System.out.println("Cannot write receipt: " + e.getMessage());
            return;
        }
        try (PrintWriter log = new PrintWriter(
                new FileWriter("receipt.txt", StandardCharsets.UTF_8, true))) {
            log.println("Thank you for choosing Profitina Laundry!");   // true = append
        } catch (IOException e) {
            System.out.println("Cannot append: " + e.getMessage());
            return;
        }

        try (BufferedReader in = new BufferedReader(
                new FileReader("receipt.txt", StandardCharsets.UTF_8))) {
            String line;
            int number = 1;
            while ((line = in.readLine()) != null) {  // null means end of file
                System.out.println(number + " | " + line);
                number++;
            }
        } catch (IOException e) {
            System.out.println("Cannot read receipt: " + e.getMessage());
        }
```

Listing 5.43 (Continued) *ReceiptWriter.java: building text with StringBuilder, writing a file, appending to it, and reading it back.*

```
    }
}
```

```
1 | WASH 3.5 kg             24,500
2 | IRON 2.0 kg             10,000
3 | EXPRESS surcharge       17,250
4 | TOTAL (Rp)              51,750
5 | Thank you for choosing Profitina Laundry!
```

Several ideas meet in this listing.

- **Building text efficiently.** Java strings are immutable, so `text = text + more` in a loop builds a brand-new string each time, copying everything so far. That is the same quadratic trap as growing an array one slot at a time (Equation (5.2)). A `StringBuilder` is a growable buffer of characters, managed exactly like the hidden array of an `ArrayList`: `append` adds to the end

cheaply, and `toString()` (called implicitly by `print`) produces the finished string once. Python's idiom, in **Listing 5.44**, is to collect the pieces in a list and join them once with `"".join(parts)`.

- **Formatting.** `%-20s` prints a string left-aligned in 20 columns, `%,10d` prints an integer right-aligned in 10 columns with thousands separators, and `%n` is the line break appropriate for the operating system.
- **Character encoding.** A file stores bytes, and an *encoding* is the rule that turns characters into bytes and back. We pass `StandardCharsets.UTF_8` explicitly to every `FileWriter` and `FileReader`, and `encoding="utf-8"` to every `open` in Python. Without it, both languages fall back on a platform default, and older Windows systems in particular use a different default, so that a file containing names such as "José" or "Müller" written on one machine could be garbled on another. Java 18 and later default to UTF-8, but saying so explicitly costs nothing and makes the program's meaning plain.
- **Reading to the end.** `while ((line = in.readLine()) != null)` is the standard Java loop for reading a text file: the assignment inside the parentheses reads a line and the comparison tests it. It stops at the end of the file, when `readLine` returns `null`.
- **Separate `try` blocks.** Writing and reading are in separate statements with their own messages, so that the user learns which step failed. The `return` in each `catch` stops the method, since there is no point in reading a receipt that could not be written.

Listing 5.44 `receipt_writer.py`: the same receipt with `"".join`, `open` modes `"w"` and `"a"`, and a `for` loop over the file.

```
def main():
    items = ["WASH 3.5 kg", "IRON 2.0 kg", "EXPRESS surcharge"]
    prices = [24500, 10000, 17250]

    parts = []                              # collect the pieces, join once
    total = 0
    for i in range(len(items)):
        parts.append(f"{items[i]:<20} {prices[i]:>10,}\n")
        total += prices[i]
    parts.append(f"{'TOTAL (Rp)':<20} {total:>10,}\n")
    text = "".join(parts)

    try:
        with open("receipt.txt", "w", encoding="utf-8") as out:
            out.write(text)                 # "w": create or overwrite
        with open("receipt.txt", "a", encoding="utf-8") as log:
            log.write("Thank you for choosing Profitina Laundry!\n")   # "a": append
    except OSError as e:
        print("Cannot write receipt:", e)
        return

    try:
        with open("receipt.txt", encoding="utf-8") as f:
            for number, line in enumerate(f, start=1):   # the loop ends at end of file
                print(str(number) + " | " + line.rstrip("\n"))
    except OSError as e:
        print("Cannot read receipt:", e)


if __name__ == "__main__":
    main()
```

```
1 | WASH 3.5 kg              24,500
2 | IRON 2.0 kg              10,000
3 | EXPRESS surcharge        17,250
4 | TOTAL (Rp)               51,750
5 | Thank you for choosing Profitina Laundry!
```

In Python, a file object opened for reading can be used directly in a `for` loop, which hands out one line at a time (each still ending with `"\n"`, which we remove with `rstrip`) and stops at the end of the file. `enumerate(f, start=1)` numbers the lines from 1. The format specification `{prices[i]:>10,}` means "right-aligned in 10 columns, with thousands separators", Python's spelling of Java's `%,10d`.

5.8.5 CSV files

A *CSV file*, for *comma-separated values*, is the simplest way of storing a table as text: one line per row, and the fields of each row separated by commas. The first line is usually a *header* naming the columns. Spreadsheets can open and save CSV files, and almost every data tool can read them, which makes CSV the most common format for exchanging tables between programs. A formal description exists as RFC 4180 [4].

The obvious way to read a CSV line is to split it at the commas. **Listing 5.45** shows that the obvious way hides two traps, and prints every field between `<` and `>` so that spaces and empty fields become visible.

Listing 5.45 `CsvSplit.java`: the two traps of splitting a CSV line in Java.

```
public class CsvSplit {
    // Shows each field between < and > so that spaces and empty fields are visible.
    static String show(String[] fields) {
        StringBuilder sb = new StringBuilder(fields.length + " fields: ");
        for (String f : fields) {
            sb.append('<').append(f).append('>');
        }
        return sb.toString();
    }

    public static void main(String[] args) {
        String line = "L-001, Sari ,WASH,3.5,";
        // split(",") silently drops the trailing empty field; -1 keeps every field.
        System.out.println("split(\",\")     -> " + show(line.split(",")));
        String[] exact = line.split(",", -1);
        System.out.println("split(\",\", -1) -> " + show(exact));
        for (int i = 0; i < exact.length; i++) {
            exact[i] = exact[i].trim();              // remove spaces around each field
        }
        System.out.println("trimmed        -> " + show(exact));

        String quoted = "L-002,\"Santoso, Budi\",IRON,2.0,no";
        System.out.println("naive split    -> " + show(quoted.split(",", -1)));
    }
}
```

```
split(",")     -> 4 fields: <L-001>< Sari ><WASH><3.5>
split(",", -1) -> 5 fields: <L-001>< Sari ><WASH><3.5><>
trimmed        -> 5 fields: <L-001><Sari><WASH><3.5><>
naive split    -> 6 fields: <L-002><"Santoso>< Budi"><IRON><2.0><no>
```

The first trap is specific to Java: `String.split(",")` *silently removes empty fields at the end* of the result. Our line has five fields, the last of them empty, but `split(",")` returns four. A program that then reads `fields[4]` crashes with `ArrayIndexOutOfBoundsException`, and only on lines whose last field happens to be empty, which makes the bug maddeningly intermittent. The cure is the second argument: `split(",", -1)` keeps every field. The fields may also carry stray spaces, which `trim()` removes.

The second trap is shared by every naive splitter: a field may itself contain a comma. The CSV convention is to put such a field in double quotation marks, `"Santoso, Budi"`, but splitting at every comma cuts the name in two and produces six fields. Handling quotes properly takes a small state machine that remembers whether it is inside quotation marks; Java's standard library does not include one, and serious Java programs use a library such as Apache Commons CSV. Python's standard library does include one, the `csv` module [5], as **Listing 5.46** shows.

Listing 5.46 `csv_split.py`: splitting by hand, and splitting correctly with the `csv` module.

```
import csv
import io


def show(fields):
    """Show each field between < and > so that spaces and empty fields are visible."""
    return str(len(fields)) + " fields: " + "".join("<" + f + ">" for f in fields)


line = "L-001, Sari ,WASH,3.5,"
exact = line.split(",")                       # Python keeps the trailing empty field
print('split(",")     -> ' + show(exact))
exact = [f.strip() for f in exact]            # remove spaces around each field
print("trimmed        -> " + show(exact))

quoted = 'L-002,"Santoso, Budi",IRON,2.0,no'
print("naive split    -> " + show(quoted.split(",")))
row = next(csv.reader(io.StringIO(quoted)))  # the csv module understands quotes
print("csv.reader     -> " + show(row))
```

```
split(",")     -> 5 fields: <L-001>< Sari ><WASH><3.5><>
trimmed        -> 5 fields: <L-001><Sari><WASH><3.5><>
naive split    -> 6 fields: <L-002><"Santoso>< Budi"><IRON><2.0><no>
csv.reader     -> 5 fields: <L-002><Santoso, Budi><IRON><2.0><no>
```

Python's `split(",")` keeps the empty last field, so the first trap does not exist there. `csv.reader` understands the quotation marks and returns five fields, with the comma inside the name preserved. (`io.StringIO` makes a string behave like a file, so that `csv.reader` can read from it; with a real file you pass the file object instead.)

For Profitina Laundry we adopt a simple, explicit rule that both languages can enforce: *no field may contain a comma*. The Java loader splits with `split(",", -1)`, the Python loader uses `csv.reader`, and any line with the wrong number of fields is rejected with a clear message. That brings us to the case study.

5.9 Profitina Laundry: Loading and Saving Orders

Profitina Laundry keeps its orders in a CSV file, `orders.csv`, which the counter staff sometimes edit by hand in a spreadsheet. Our task is to load the file into a list of `Order` objects, reject every invalid line with a message that says exactly what is wrong and where, keep going after a bad line, compute prices, and save the valid orders to a clean file that the program can read back. The plan is drawn in **Figure 5.7**.

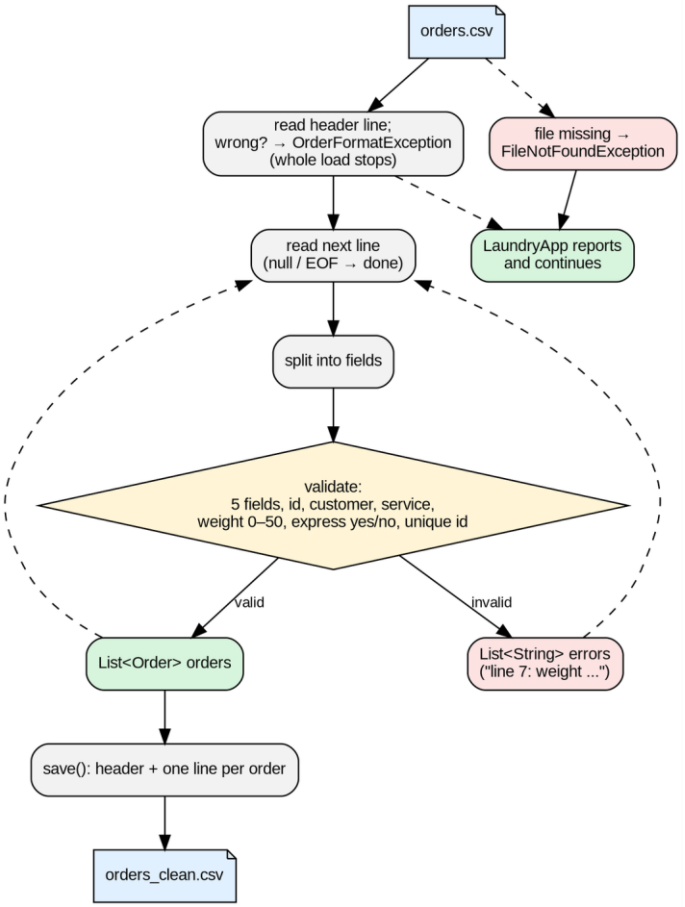


Figure 5.7 The order-loading pipeline. A bad line becomes a message in the error list and loading continues; a missing file or a wrong header stops the load and is reported by the application.

Two kinds of failure are treated differently, and the distinction is the heart of the design. A *bad line* is a local problem: the rest of the file may be perfectly good, so we record the problem and continue. A *missing file*, or a file whose header shows that it is not an orders file at all, is a global problem: there is nothing sensible to load, so the loader stops and throws, and the application decides what to tell the user.

Here is the file. It has been deliberately damaged in seven different ways, and it contains a blank line:

```
id,customer,service,weight_kg,express
L-001,Sari,WASH,3.5,no
L-002,Budi Santoso,WASH_IRON,5.0,yes
L-003,Rina,IRON,2,no
L-004,,WASH,4.0,no
L-005,Tono,DRY_CLEAN,1.5,no
L-006,Wati,WASH,abc,no
L-007,Agus,WASH,-3,no
L-002,Dewi,WASH,2.0,no
L-008,Joko,IRON,1.2

L-009,Lestari,WASH_IRON,7.25,maybe
L-010,Hendra,WASH,12,yes
```

The laundry offers three services, charged per kilogram: `WASH` at Rp 7,000, `IRON` at Rp 5,000 and `WASH_IRON` at Rp 10,000. An express order costs 50% more. For an order of weight w kilograms at rate r rupiah per kilogram, the price is

$$p = \begin{cases} \text{round}(w \cdot r) & \text{normal order,} \\ \text{round}(w \cdot r) + \lfloor \text{round}(w \cdot r)/2 \rfloor & \text{express order.} \end{cases} \tag{5.4}$$

We compute in whole rupiah with integers, because amounts of money should never be stored in floating-point variables longer than necessary: $0.1 + 0.2$ is not exactly 0.3 in binary floating point. For example, by Equation (5.4), Sari's 3.5 kg wash costs $\text{round}(3.5 \times 7000) = 24{,}500$, and Budi's 5 kg express wash-and-iron costs $50{,}000 + 25{,}000 = 75{,}000$.

The validation rules are: exactly five fields; a non-empty id and customer name; a known service; a weight that is a number greater than 0 and at most 50 kg; an express flag of `yes` or `no`; and an id that has not been used on an earlier line.

5.9.1 The `Order` class

Listing 5.47 is the class that represents one order. It knows its price and how to write itself as a CSV line.

Listing 5.47 `Order.java`: one laundry order, with its price and its CSV form.

```
import java.util.Locale;

public class Order {
    // Parallel arrays: SERVICES[i] costs RATES[i] rupiah per kilogram.
    static final String[] SERVICES = {"WASH", "IRON", "WASH_IRON"};
    static final long[] RATES = {7000, 5000, 10000};

    private final String id;
    private final String customer;
    private final String service;
    private final double weightKg;
    private final boolean express;

    public Order(String id, String customer, String service, double weightKg, boolean
express) {
        this.id = id;
        this.customer = customer;
        this.service = service;
        this.weightKg = weightKg;
        this.express = express;
    }

    /** Returns the rate for a service code, or -1 if the code is unknown. */
    static long rateFor(String service) {
        for (int i = 0; i < SERVICES.length; i++) {
            if (SERVICES[i].equals(service)) {
                return RATES[i];
            }
        }
        return -1;
    }

    public long price() {
        long base = Math.round(weightKg * rateFor(service));
        return express ? base + base / 2 : base;       // express costs 50% more
    }

    public String getId() { return id; }
    public String getService() { return service; }
    public double getWeightKg() { return weightKg; }

    public String toCsv() {
        return id + "," + customer + "," + service + "," + weightKg + ","
                + (express ? "yes" : "no");
    }
```

Listing 5.47 (Continued) *Order.java: one laundry order, with its price and its CSV form.*

```
    @Override
    public String toString() {
        return String.format(Locale.US, "%-6s %-13s %-9s %5.2f kg %-3s Rp %,9d",
                id, customer, service, weightKg, express ? "exp" : "", price());
    }
}
```

The services and their rates are kept in two *parallel arrays*: `SERVICES[i]` costs `RATES[i]` per kilogram. They are arrays, not lists, because the set of services is fixed when the program is written. `rateFor` searches `SERVICES` with an ordinary loop and returns −1 for an unknown code, the same convention as `indexOf`. `price()` implements Equation (5.4): `Math.round` rounds the product to the nearest whole rupiah and returns a `long`, and `base / 2` is integer division. `toString` uses `Locale.US` so that the thousands separator is a comma on every computer; an Indonesian locale would print `24.500` instead, which is correct in Indonesia but would make the output differ from machine to machine.

The Python class, **Listing 5.48**, uses a dictionary for the rates. A dictionary maps each key to a value directly and is Python's natural tool for this job; Chapter 9 studies Java's equivalent, the `Map`.

Listing 5.48 `order.py`: the same class in Python, with a dictionary of rates.

```
# A dictionary plays the part of Java's two parallel arrays.
RATES = {"WASH": 7000, "IRON": 5000, "WASH_IRON": 10000}


def rate_for(service):
    """Return the rate for a service code, or -1 if the code is unknown."""
    return RATES.get(service, -1)


class Order:
    def __init__(self, order_id, customer, service, weight_kg, express):
        self.order_id = order_id
        self.customer = customer
        self.service = service
        self.weight_kg = weight_kg
        self.express = express

    def price(self):
        base = round(self.weight_kg * rate_for(self.service))
        return base + base // 2 if self.express else base   # express costs 50% more

    def to_csv(self):
        return ",".join([self.order_id, self.customer, self.service,
                         str(self.weight_kg), "yes" if self.express else "no"])

    def __str__(self):
        return (f"{self.order_id:<6} {self.customer:<13} {self.service:<9} "
                f"{self.weight_kg:5.2f} kg {'exp' if self.express else '':<3} "
                f"Rp {self.price():9,}")
```

5.9.2 The exception

A bad line is described by a custom checked exception that records the line number, **Listing 5.49**. It extends `Exception`, not `RuntimeException`, because invalid data from outside the program is an expected outcome that every caller must plan for.

Listing 5.49 `OrderFormatException.java`: a checked exception for one invalid line.

```
/** A checked exception: one line of an orders file is not a valid order. */
public class OrderFormatException extends Exception {
    private final int lineNumber;

    public OrderFormatException(int lineNumber, String message) {
        this(lineNumber, message, null);
    }

    public OrderFormatException(int lineNumber, String message, Throwable cause) {
        super("line " + lineNumber + ": " + message, cause);
        this.lineNumber = lineNumber;
    }

    public int getLineNumber() {
        return lineNumber;
    }
}
```

The first constructor passes its work to the second with `this(...)`, supplying `null` for the cause, so that the message is built in only one place. Every message begins with the line number, so the user can go straight to the problem in a spreadsheet. **Listing 5.50** is the Python counterpart.

Listing 5.50 `order_errors.py`: the same exception in Python.

```
class OrderFormatError(Exception):
    """One line of an orders file is not a valid order."""

    def __init__(self, line_number, message):
        super().__init__("line " + str(line_number) + ": " + message)
        self.line_number = line_number
```

5.9.3 Loading and saving

Listing 5.51 is the heart of the case study.

Listing 5.51 `OrderCsv.java`: parsing and validating one line, loading a whole file, and saving.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;

public class OrderCsv {
    static final String HEADER = "id,customer,service,weight_kg,express";

    /** Turns one CSV line into an Order, or explains exactly why it cannot. */
    static Order parseLine(String line, int lineNumber) throws OrderFormatException {
        String[] f = line.split(",", -1);
        if (f.length != 5) {
            throw new OrderFormatException(lineNumber, "expected 5 fields, found " +
f.length);
        }
        for (int i = 0; i < f.length; i++) {
            f[i] = f[i].trim();
        }
        if (f[0].isEmpty()) {
            throw new OrderFormatException(lineNumber, "order id is empty");
        }
        if (f[1].isEmpty()) {
            throw new OrderFormatException(lineNumber, "customer name is empty");
        }
        if (Order.rateFor(f[2]) < 0) {
            throw new OrderFormatException(lineNumber, "unknown service \"" + f[2] + "\"");
        }
        double kg;
        try {
            kg = Double.parseDouble(f[3]);
        } catch (NumberFormatException nfe) {
            throw new OrderFormatException(lineNumber,
                    "weight \"" + f[3] + "\" is not a number", nfe);
        }
        if (!(kg > 0 && kg <= 50)) {
            throw new OrderFormatException(lineNumber, "weight " + f[3] + " is outside 0-50
kg");
        }
        if (!f[4].equals("yes") && !f[4].equals("no")) {
            throw new OrderFormatException(lineNumber,
                    "express must be yes or no, found \"" + f[4] + "\"");
        }
        return new Order(f[0], f[1], f[2], kg, f[4].equals("yes"));
    }
```

Listing 5.51 (Continued) *OrderCsv.java: parsing and validating one line, loading a whole file, and saving.*

```
    /**
     * Loads every valid order; each invalid line becomes one message in errors.
     * A missing file or a wrong header stops the whole load with an exception.
     */
    static List<Order> load(String fileName, List<String> errors)
            throws IOException, OrderFormatException {
        List<Order> orders = new ArrayList<>();
        List<String> seenIds = new ArrayList<>();
        try (BufferedReader in = new BufferedReader(
                new FileReader(fileName, StandardCharsets.UTF_8))) {
            String header = in.readLine();
            if (header == null || !header.trim().equals(HEADER)) {
                throw new OrderFormatException(1, "header must be " + HEADER);
            }
            int lineNumber = 1;
            String line;
            while ((line = in.readLine()) != null) {
                lineNumber++;
                if (line.isBlank()) {
                    continue;                              // tolerate empty lines
                }
                try {
                    Order order = parseLine(line, lineNumber);
                    if (seenIds.contains(order.getId())) {
                        throw new OrderFormatException(lineNumber, "duplicate id " +
order.getId());
                    }
                    seenIds.add(order.getId());
                    orders.add(order);
                } catch (OrderFormatException e) {
                    errors.add(e.getMessage());            // record it and keep going
                }
            }
        }
        return orders;
    }
```

Listing 5.51 (Continued) *OrderCsv.java: parsing and validating one line, loading a whole file, and saving.*

```
    static void save(String fileName, List<Order> orders) throws IOException {
        try (PrintWriter out = new PrintWriter(new FileWriter(fileName,
StandardCharsets.UTF_8))) {
            out.println(HEADER);
            for (Order o : orders) {
                out.println(o.toCsv());
            }
        }
    }
}
```

Follow a line through `parseLine`. It is split with `split(",", -1)`, so that an empty last field still counts, and the number of fields is checked *before* any field is used, so that a short line can never cause an `ArrayIndexOutOfBoundsException`. Each field is trimmed. Then the rules are checked one at a time, and the first rule that fails throws an `OrderFormatException` with a specific message. The weight is the only field that needs a conversion that can fail, so `NumberFormatException` is caught there and *wrapped* as the cause of an `OrderFormatException`, exactly as in **Listing 5.35**. The test `!(kg > 0 && kg <= 50)` is written that way round on purpose: it also rejects the special value "not a number" (`NaN`), which

`Double.parseDouble("NaN")` accepts and for which every comparison is false. Only a line that survives every check becomes an `Order`.

`load` declares two checked exceptions, `IOException` and `OrderFormatException`, and the header comment says when each is thrown. The file is opened with `try`-with-resources, so it is closed whatever happens. The first line must be the header; if not, the method throws, and the load stops. Then each remaining line goes through its own small `try`: a successful parse is checked for a duplicate id and added to the list; an `OrderFormatException` is caught *inside the loop*, its message is added to the `errors` list supplied by the caller, and the loop goes on to the next line. The position of that inner `try` is the design decision of **Figure 5.7** expressed in code. The duplicate test reuses the exception by throwing it inside the same `try`, so that all bad lines are reported in one uniform way. The list `seenIds` is searched with `contains`, which examines each element in turn; Chapter 9 introduces the `Set`, which answers the same question much faster for large files.

`save` writes the header and one line per order with `try`-with-resources. It declares `IOException` and leaves the decision to the caller.

The Python module, **Listing 5.52**, has the same structure. It reads the file through `csv.reader`, whose `line_num` attribute gives the number of the line just read, and opens files with `newline=""`, as the `csv` documentation requires. `parse_row` receives a list of fields from `csv.reader`, and `parse_line` wraps it for a single line of text.

Listing 5.52 `order_csv.py`: the same loading, validation and saving in Python.

```
import csv

from order import Order, rate_for
from order_errors import OrderFormatError

HEADER = ["id", "customer", "service", "weight_kg", "express"]


def parse_row(f, line_number):
    """Turn one CSV row (a list of strings) into an Order, or explain why not."""
    if len(f) != 5:
        raise OrderFormatError(line_number, "expected 5 fields, found " + str(len(f)))
    f = [field.strip() for field in f]
    if f[0] == "":
        raise OrderFormatError(line_number, "order id is empty")
    if f[1] == "":
        raise OrderFormatError(line_number, "customer name is empty")
    if rate_for(f[2]) < 0:
        raise OrderFormatError(line_number, 'unknown service "' + f[2] + '"')
    try:
        kg = float(f[3])
    except ValueError as ve:
        raise OrderFormatError(line_number, 'weight "' + f[3] + '" is not a number') from
ve
    if not (0 < kg <= 50):
        raise OrderFormatError(line_number, "weight " + f[3] + " is outside 0-50 kg")
    if f[4] not in ("yes", "no"):
        raise OrderFormatError(line_number, 'express must be yes or no, found "' + f[4] +
'"')
    return Order(f[0], f[1], f[2], kg, f[4] == "yes")


def parse_line(line, line_number):
    return parse_row(line.split(","), line_number)


def load(file_name, errors):
    """Load every valid order; each invalid line becomes one message in errors.
```

Listing 5.52 (Continued) *order_csv.py: the same loading, validation and saving in Python.*

```
    Raises:
        FileNotFoundError: the file does not exist.
        OrderFormatError: the header is wrong, so this is not an orders file.
    """
    orders = []
    seen_ids = []
    with open(file_name, newline="", encoding="utf-8") as f:
        reader = csv.reader(f)
        header = next(reader, None)
        if header is None or [h.strip() for h in header] != HEADER:
            raise OrderFormatError(1, "header must be " + ",".join(HEADER))
        for row in reader:
            line_number = reader.line_num
            if row == [] or all(field.strip() == "" for field in row):
                continue                          # tolerate empty lines
            try:
                order = parse_row(row, line_number)
                if order.order_id in seen_ids:
                    raise OrderFormatError(line_number, "duplicate id " + order.order_id)
                seen_ids.append(order.order_id)
                orders.append(order)
            except OrderFormatError as e:
                errors.append(str(e))             # record it and keep going
    return orders


def save(file_name, orders):
    with open(file_name, "w", newline="", encoding="utf-8") as f:
        f.write(",".join(HEADER) + "\n")
        for o in orders:
            f.write(o.to_csv() + "\n")
```

5.9.4 The application

Finally, **Listing 5.53** ties everything together. `loadAndReport` is where the decisions about global failures are made: it catches `FileNotFoundException`, `OrderFormatException` and any other `IOException`, each with a message suited to it, most specific first. `main` loads `orders.csv`, adds a new order typed at the counter, prints every order with its price and the total revenue, saves the valid orders to `orders_clean.csv`, and reads that file back to prove that what we write, we can read. Then, to show the global failures, it tries a file that does not exist and a file that is not an orders file at all.

Listing 5.53 `LaundryApp.java`: loading, reporting, pricing, saving and reloading the orders.

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

public class LaundryApp {
    /** Loads a file and reports what happened; returns null if nothing could be loaded. */
    static List<Order> loadAndReport(String fileName) {
        List<String> errors = new ArrayList<>();
        try {
            List<Order> orders = OrderCsv.load(fileName, errors);
            System.out.println("Loaded " + orders.size() + " order(s) from " + fileName
                    + ", rejected " + errors.size() + " line(s).");
            for (String error : errors) {
                System.out.println("  rejected " + error);
            }
            return orders;
        } catch (FileNotFoundException e) {
            System.out.println("No such file: " + fileName + ". Starting with no orders.");
        } catch (OrderFormatException e) {
            System.out.println("Not an orders file (" + fileName + ", " + e.getMessage() +
").");
        } catch (IOException e) {
            System.out.println("Could not read " + fileName + ": " + e.getMessage());
        }
        return null;
    }

    public static void main(String[] args) {
        List<Order> orders = loadAndReport("orders.csv");
        if (orders == null) {
            return;
        }
        try {
            orders.add(OrderCsv.parseLine("L-011,Maya,IRON,3.0,yes", 0));
        } catch (OrderFormatException e) {
            System.out.println("Cannot add order: " + e.getMessage());
        }

        long revenue = 0;
        for (Order o : orders) {
            System.out.println(o);
            revenue += o.price();
        }
        System.out.println(String.format(Locale.US, "Revenue: Rp %,d", revenue));
```

Listing 5.53 (Continued) *LaundryApp.java: loading, reporting, pricing, saving and reloading the orders.*

```
        try {
            OrderCsv.save("orders_clean.csv", orders);
            System.out.println("Saved " + orders.size() + " order(s) to orders_clean.csv");
        } catch (IOException e) {
            System.out.println("Could not save: " + e.getMessage());
        }
        loadAndReport("orders_clean.csv");               // round trip: read our own file
back
        loadAndReport("missing.csv");
        loadAndReport("notes.txt");
    }
}
```

```
Loaded 4 order(s) from orders.csv, rejected 7 line(s).
  rejected line 5: customer name is empty
  rejected line 6: unknown service "DRY_CLEAN"
  rejected line 7: weight "abc" is not a number
  rejected line 8: weight -3 is outside 0-50 kg
  rejected line 9: duplicate id L-002
  rejected line 10: expected 5 fields, found 4
  rejected line 12: express must be yes or no, found "maybe"
L-001  Sari          WASH       3.50 kg     Rp    24,500
L-002  Budi Santoso  WASH_IRON  5.00 kg exp Rp    75,000
L-003  Rina          IRON       2.00 kg     Rp    10,000
L-010  Hendra        WASH      12.00 kg exp Rp   126,000
L-011  Maya          IRON       3.00 kg exp Rp    22,500
Revenue: Rp 258,000
Saved 5 order(s) to orders_clean.csv
Loaded 5 order(s) from orders_clean.csv, rejected 0 line(s).
No such file: missing.csv. Starting with no orders.
Not an orders file (notes.txt, line 1: header must be
id,customer,service,weight_kg,express).
```

Check the output against the input file. Line 1 is the header and lines 2, 3 and 4 are valid. Line 5 has an empty customer. Line 6 names a service the laundry does not offer. Line 7's weight is not a number. Line 8's weight is negative. Line 9 reuses the id `L-002`. Line 10 has only four fields. Line 11 is blank and is skipped without comment. Line 12 has `maybe` for the express flag. Line 13 is valid. Four orders survive, and with Maya's order added at the counter there are five, whose prices by Equation (5.4) sum to $24{,}500 + 75{,}000 + 10{,}000 + 126{,}000 + 22{,}500 = 258{,}000$ rupiah. The saved file is:

```
id,customer,service,weight_kg,express
L-001,Sari,WASH,3.5,no
L-002,Budi Santoso,WASH_IRON,5.0,yes
L-003,Rina,IRON,2.0,no
L-010,Hendra,WASH,12.0,yes
L-011,Maya,IRON,3.0,yes
```

It loads back with no errors, which is the round-trip test every save routine deserves. Notice that the saved weights are written as `2.0` and `12.0`, the way `Double.toString` and Python's `str` write a `double`; the file is still valid, since our loader reads both forms.

Listing 5.54 is the Python application. The only structural difference is the use of `else` after the `except` clauses: the success report runs only if `load` raised nothing.

Listing 5.54 `laundry_app.py`: the Python application.

```
import order_csv
from order_errors import OrderFormatError


def load_and_report(file_name):
    """Load a file and report what happened; return None if nothing could be loaded."""
    errors = []
    try:
        orders = order_csv.load(file_name, errors)
    except FileNotFoundError:
        print("No such file: " + file_name + ". Starting with no orders.")
    except OrderFormatError as e:
        print("Not an orders file (" + file_name + ", " + str(e) + ").")
    except OSError as e:
        print("Could not read " + file_name + ": " + str(e))
    else:
        print("Loaded " + str(len(orders)) + " order(s) from " + file_name
              + ", rejected " + str(len(errors)) + " line(s).")
        for error in errors:
            print("  rejected " + error)
        return orders
    return None


def main():
    orders = load_and_report("orders.csv")
    if orders is None:
        return
    try:
        orders.append(order_csv.parse_line("L-011,Maya,IRON,3.0,yes", 0))
    except OrderFormatError as e:
        print("Cannot add order: " + str(e))

    revenue = 0
    for o in orders:
        print(o)
        revenue += o.price()
    print(f"Revenue: Rp {revenue:,}")
```

Listing 5.54 (Continued) *laundry_app.py: the Python application.*

```
    try:
        order_csv.save("orders_clean.csv", orders)
        print("Saved " + str(len(orders)) + " order(s) to orders_clean.csv")
    except OSError as e:
        print("Could not save: " + str(e))
    load_and_report("orders_clean.csv")           # round trip: read our own file back
    load_and_report("missing.csv")
    load_and_report("notes.txt")


if __name__ == "__main__":
    main()
```

```
Loaded 4 order(s) from orders.csv, rejected 7 line(s).
  rejected line 5: customer name is empty
  rejected line 6: unknown service "DRY_CLEAN"
  rejected line 7: weight "abc" is not a number
  rejected line 8: weight -3 is outside 0-50 kg
  rejected line 9: duplicate id L-002
  rejected line 10: expected 5 fields, found 4
  rejected line 12: express must be yes or no, found "maybe"
L-001  Sari          WASH       3.50 kg     Rp    24,500
L-002  Budi Santoso  WASH_IRON  5.00 kg exp Rp    75,000
L-003  Rina          IRON       2.00 kg     Rp    10,000
L-010  Hendra        WASH      12.00 kg exp Rp   126,000
L-011  Maya          IRON       3.00 kg exp Rp    22,500
Revenue: Rp 258,000
Saved 5 order(s) to orders_clean.csv
Loaded 5 order(s) from orders_clean.csv, rejected 0 line(s).
No such file: missing.csv. Starting with no orders.
Not an orders file (notes.txt, line 1: header must be
id,customer,service,weight_kg,express).
```

The two outputs are identical line for line, and the two programs write byte-identical `orders_clean.csv` files. To run the Java version, compile the four classes together and start `LaundryApp`:

```
javac Order.java OrderFormatException.java OrderCsv.java LaundryApp.java
java LaundryApp
```

and for Python, with `order.py`, `order_errors.py` and `order_csv.py` in the same folder, `python3 laundry_app.py`. On Windows, type `python` instead of `python3`. The `README.md` in the chapter's code folder gives the details for every program in the chapter.

5.10 Profitina in Practice

Profitina, the AI visibility platform at app.profitina.com, spends much of its life doing on a large scale what Section 5.9 does on a small one: taking data it does not control, checking it, and refusing politely what it cannot use.

Lists whose size nobody knows in advance. When Profitina scans a website to measure how ready it is to be cited by AI answer engines, it does not know beforehand how many pages it will find, how many structured-data blocks a page contains, or how many signals will need explaining. Every one of these collections is a growable list, Python's own `list` on the FastAPI back end, for exactly the reason given in Section 5.3.1. Fixed-size arrays appear where sizes really are fixed, for example in numerical work on vectors of scores.

CSV at the edges. The historical analytics dashboard offers a CSV export of each brand's weekly score trend, so that a brand owner can open it in a spreadsheet, and the self-service data export required by Indonesia's personal data protection law (UU PDP 27/2022) gives users their own data in portable form. Writing such files is Section 5.8.4 and 5.8.5 applied with care: an explicit UTF-8 encoding, a header row, and fields that are quoted properly when they contain commas.

Failures that are reported, not hidden. The platform's AI features run entirely on a local model (Qwen2.5, served by Ollama) so that user data never leaves the server. A local model can be unavailable: the process may be restarting, or the machine may be busy. When that happens the application *says so*, rather than silently serving a canned answer as if it came from the model. That is the principle of Section 5.5 at the scale of a product: an exception is information, and swallowing it deceives the user. In the same spirit, a staged release reports success only after the

newly deployed version has answered its health check, so a failure in deployment cannot masquerade as a success.

Validation before trust. Every link posted in Profitina's community is screened before it is shown, and a post that fails to send is not lost: the composer restores the text so that the user can try again. Both are the pattern of Section 5.9, which keeps what is valid, explains what is not, and never lets one bad item bring down everything else.

The rule behind all of these is the house rule of the whole Profitina group, and it applies to the laundry program too: the software advises, the user decides. A program can only give good advice if it tells the truth about what went wrong.

5.11 Common Mistakes, and How to Avoid Them

Looping one step too far. `for (int i = 0; i <= a.length; i++)` visits the non-existent index `a.length` and throws `ArrayIndexOutOfBoundsException`. *Avoid it:* the condition is always `i < a.length` (or `i < list.size()`); better still, use the enhanced `for` or Python's `for x in lst` when the index is not needed.

Hard-coding a length. `i < 5` works until the array grows to six. *Avoid it:* use `.length`, `size()` or `len()`.

Confusing `length`, `length()` and `size()`. Arrays have the field `length`; strings the method `length()`; lists the method `size()`. *Avoid it:* remember "array field, string method, list size", and let the compiler catch the rest.

Copying a reference and calling it a copy. `int[] b = a;` or `b = a` makes an alias. *Avoid it:* use `Arrays.copyOf` in Java, `a.copy()` or `list(a)` in Python, and draw the arrows as in **Figure 5.1** whenever you are unsure.

Comparing arrays or lists with the wrong operator. In Java `==` compares references; use `Arrays.equals` for arrays and `equals` for lists. In Python `==` compares contents and `is` compares identity.

`remove(3)` on a list of `Integer`. It removes the element at index 3, not the value 3. *Avoid it:* write `remove(Integer.valueOf(3))` for the value.

Expecting `-1` from Python's `index`. `lst.index(x)` raises `ValueError` when `x` is absent. *Avoid it:* test `x in lst` first, or catch the exception.

The leftover line break after `nextInt()`. A `nextLine()` straight after `nextInt()` returns `""`. *Avoid it:* call `nextLine()` once to finish the number's line, as in **Listing 5.15**, or read every line with `nextLine()` and convert it with `Integer.parseInt`.

Retrying a failed `nextInt()` without discarding the token. The bad token stays in the input and the loop never ends. *Avoid it:* call `s.next()` in the `catch` block, as in **Listing 5.23**.

Catching `Exception` first, or catching it everywhere. A general net before a specific one does not compile in Java and, in Python, silently makes the specific clause dead code. A general net in the wrong place hides bugs. *Avoid it:* specific nets first, and the general one only at the top level of a loop or program, where continuing is truly right.

The empty `catch` block. `catch (IOException e) { }` or `except: pass` throws away the evidence. *Avoid it:* at the very least print the message; better, handle the problem or let it propagate.

Catching an unchecked exception instead of fixing the bug. Wrapping `arr[i]` in `try`/`catch (ArrayIndexOutOfBoundsException e)` to hide an off-by-one error only moves the bug. *Avoid it:* fix the index.

Declaring `throws` on `main` to silence the compiler. The checked exception then crashes the program with a stack trace. *Avoid it:* catch checked exceptions in the method that knows what to tell the user.

Using a variable from inside a `try` block after the block. It does not exist there. *Avoid it:* do the work inside the `try`, or declare the variable before it.

Forgetting to close a file. Written data may never reach the disk, and handles run out. *Avoid it:* `try`-with-resources in Java, `with` in Python, every time.

Relying on the default character encoding. A file written on one computer is garbled on another. *Avoid it:* pass `StandardCharsets.UTF_8` or `encoding="utf-8"` explicitly.

`split(",")` in Java losing trailing empty fields. *Avoid it:* `split(",", -1)`, and check the number of fields before using any of them.

A method whose name lies. `isExceeded` in **Listing 5.29** returns `true` when the value is *not* exceeded. *Avoid it:* name a method for what it returns; a method that only checks and throws can be called `checkGrayscale` and return nothing.

Using exceptions for ordinary control flow. Throwing to leave a loop, or to return a normal result, is slow and misleading. *Avoid it:* keep exceptions for errors.

5.12 Chapter Summary

- An **array** is a fixed-length row of slots of one type, indexed from 0 to $n - 1$. By Equation (5.1), any slot can be reached directly. `new T[n]` fills it with default values; `{a, b, c}` is the shorthand. Traverse with `i < a.length` or with the enhanced `for`. Index n throws `ArrayIndexOutOfBoundsException` (`IndexError` in Python).
- An array variable holds a **reference**. Assignment makes an alias; `Arrays.copyOf` and `System.arraycopy` make copies. An array cannot grow; making a bigger array and copying is the only way.
- `main(String[] args)` receives the **command-line arguments**; Python has `sys.argv`, where index 0 is the script.
- A **list** (`ArrayList<T>`, Python `list`) grows by itself. Its core operations are add/append, get/`[i]`, set, size/`len`, contains/`in`, indexOf/`index`, and remove by position or by value (**Table 5.1**). Growing geometrically makes appends cheap on average (Equation (5.3)), whereas growing by one slot costs quadratically (Equation (5.2)). Java lists of numbers need wrapper types such as `Integer`.
- **Input**: Java's `Scanner` reads lines (`nextLine`) and typed tokens (`nextInt`, `nextDouble`); beware the leftover line break. Python's `input()` reads a line; convert with `int()` or `float()`. Redirect a file into standard input with `<` for repeatable runs.
- An **exception** is an object describing a failure. It is **thrown** (`throw`, `raise`), **propagates** back through the calls, and is **caught** by the first matching `catch`/`except`. Read Java stack traces top-down for the type and look for your own class; read Python tracebacks bottom-up.
- Exceptions form a **class hierarchy**; a `catch` for a class also catches its subclasses, so order the clauses from specific to general. `finally` always runs. Python adds `else`.

- In Java, subclasses of `RuntimeException` are **unchecked**, usually programming mistakes to be fixed; other exceptions such as `IOException` are **checked** and must be **caught or declared** with `throws`. Python has no checked exceptions; document them in docstrings.
- **Custom exceptions** extend `Exception` (checked) or `RuntimeException` (unchecked) in Java, `Exception` or a subclass in Python; they carry messages, extra data and a **cause** (`super(message, cause)`, `raise ... from ...`).
- **Files**: streams carry bytes, readers and writers carry characters, and buffering makes both efficient. Always close what you open, with **`try`-with-resources** or `with`. Specify UTF-8. Build long text with `StringBuilder` or `"".join`.
- **CSV**: one row per line, fields separated by commas, usually with a header. Use `split(",", -1)` in Java, the `csv` module in Python, and validate every field.
- **Profitina Laundry** now loads orders from CSV, rejects each bad line with its line number and reason, continues past it, stops cleanly on a missing or foreign file, and saves a clean file that reloads without errors.

5.13 Exercises

Exercise 5.1. (a) What happens if, after creating `String[] fruitArray = new String[5]`, we write `fruitArray[5] = "durian";`? Why? (b) Write the statement that creates an array of `int`s of size ten, in Java and in Python. (c) What values does `new boolean[4]` contain, and what does `new String[2]` contain?

Exercise 5.2. Write a method that returns the tenth fruit of an `ArrayList<String>` (a Python function for a `list`). Bear in mind that there may not be ten fruit in the list, and handle that case without any exception. Test it on lists of 3 and 10 fruit.

Exercise 5.3. For each situation, choose an array or a list and justify the choice in one sentence: (a) the seven daily totals of one week; (b) the orders received today; (c) a 3×3 tic-tac-toe board; (d) the customers who answered a survey; (e) the 256 counts of a grayscale histogram; (f) the lines of a text file of unknown length.

Exercise 5.4. Without using `ArrayList` or `append`, write a method that appends a value to an array by making a new array one slot longer. Append 1000 values and count how many element copies were made. Then implement the doubling strategy of Section 5.3.3, keeping a separate size, and count again. Compare both counts with Equation (5.2) and Equation (5.3).

Exercise 5.5. Without running it, state exactly what this Java method prints for `f(0)` and for `f(1)`, and what it returns:

```
static int f(int x) {
    try {
        System.out.println("A");
        int y = 10 / x;
        System.out.println("B");
        return y;
    } catch (ArithmeticException e) {
        System.out.println("C");
        return -1;
    } finally {
        System.out.println("D");
    }
}
```

Exercise 5.6. Classify each Java exception as checked or unchecked, and say whether it usually signals a programming error or an outside problem: `NullPointerException`,

`FileNotFoundException`, `NumberFormatException`, `IOException`, `ClassCastException`, `InputMismatchException`, `ArithmeticException`, `OrderFormatException` (Section 5.9).

Exercise 5.7. Write a method `getSomeNumbers(int first, int second)` that throws `IllegalArgumentException` with the message "The numbers you've entered are not acceptable." when `first + second >= 20`, and otherwise prints "Your numbers are fine. Well done." Call it from `main` for the pairs (3,4), (15,5), (−30,45) and (9,10), catching the exception. Write the Python version with `ValueError`.

Exercise 5.8. Write a program that reads whitespace-separated tokens from standard input until the token `99`, skips every token that is not an integer, and finally prints how many integers were read, how many tokens were skipped, and the average of the integers to two decimal places. It must not crash if the input ends without `99` or if no integers are read. Use the input `4 x 8 / 15 3.5 -2 / six 99 1000` (three lines, slashes marking the line breaks).

Exercise 5.9. Find all the mistakes in this program and say which one the compiler reports:

```
public class Buggy {
    public static void main(String[] args) {
        String[] prices = {"7000", "5000", "ten thousand"};
        int total = 0;
        for (int i = 0; i <= prices.length; i++) {
            try {
                total += Integer.parseInt(prices[i]);
            } catch (Exception e) {
            } catch (NumberFormatException e) {
                System.out.println("bad price " + prices[i]);
            }
        }
        System.out.println("total = " + total);
    }
}
```

Exercise 5.10. Profitina Laundry sells prepaid cards. Write a class `PrepaidCard` with an owner and a balance in rupiah, and a method `pay(long amount)` that throws `IllegalArgumentException` for a non-positive amount and a custom *checked* exception `InsufficientBalanceException` when the balance is too small, carrying how much is missing. A failed payment must leave the balance untouched. Test with a balance of 50,000 and the bills 24,500, 30,000, 20,000 and −5. Write the Python version too.

Exercise 5.11. Using `OrderCsv` (or `order_csv`) from Section 5.9, load `orders.csv`, count the orders, the kilograms and the revenue per service, and write them to `summary.csv` with the header `service,orders,kg,revenue`.

Exercise 5.12. A first version of **Listing 5.23** handled bad input by writing `s = new Scanner(System.in);` in the `catch` block instead of `s.next();`. Explain why this may seem to work at the keyboard but fails when the input is redirected from a file.

Exercise 5.13. Write a program `WordCount` (Python `word_count.py`) that takes a file name as its only command-line argument and prints the number of lines, words and characters in the file (not counting line breaks). With no argument it prints a usage message; for a missing file it prints "no such file:" and the name. Try it on `notes.txt`.

5.14 Solutions to the Exercises

Solution 5.1. (a) The program throws `ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5`. An array of length 5 has the indexes 0 to 4 (**Figure 5.1**); index 5 would be

a sixth slot, and a Java array never grows. (b) Java: `int[] tenInts = new int[10];` Python: `ten_ints = [0] * 10`. (c) `new boolean[4]` holds four `false`s and `new String[2]` holds two `null`s: every new Java array is filled with the default value of its element type.

Solution 5.2. The tenth element has index 9, and it exists only when the size is at least 10, so we test the size before calling `get`. **Listing 5.55** and **Listing 5.56** give the solution.

Listing 5.55 `TenthFruit.java`: the tenth fruit, if there is one.

```
import java.util.ArrayList;

public class TenthFruit {
    static String tenthFruit(ArrayList<String> fruit) {
        if (fruit.size() >= 10) {                // the 10th fruit lives at index 9
            return fruit.get(9);
        }
        return "(only " + fruit.size() + " fruit, so there is no 10th)";
    }

    public static void main(String[] args) {
        ArrayList<String> fruit = new ArrayList<>();
        String[] names = {"apel", "belimbing", "ceri", "duku", "durian",
                          "jambu", "jeruk", "kedondong", "kelengkeng", "mangga"};
        for (String name : names) {
            fruit.add(name);
            if (fruit.size() == 3 || fruit.size() == 10) {
                System.out.println(fruit.size() + " fruit: " + tenthFruit(fruit));
            }
        }
    }
}
```

```
3 fruit: (only 3 fruit, so there is no 10th)
10 fruit: mangga
```

Listing 5.56 `tenth_fruit.py`: the same function in Python.

```
def tenth_fruit(fruit):
    if len(fruit) >= 10:                     # the 10th fruit lives at index 9
        return fruit[9]
    return "(only " + str(len(fruit)) + " fruit, so there is no 10th)"


def main():
    fruit = []
    names = ["apel", "belimbing", "ceri", "duku", "durian",
             "jambu", "jeruk", "kedondong", "kelengkeng", "mangga"]
    for name in names:
        fruit.append(name)
        if len(fruit) == 3 or len(fruit) == 10:
            print(str(len(fruit)) + " fruit: " + tenth_fruit(fruit))


if __name__ == "__main__":
    main()
```

```
3 fruit: (only 3 fruit, so there is no 10th)
10 fruit: mangga
```

Solution 5.3. (a) Array: exactly seven values, always. (b) List: nobody knows how many orders will come. (c) Two-dimensional array: the shape is fixed and grid-like. (d) List: the number of respondents is unknown. (e) Array (`int[256]`): the number of possible gray levels is fixed. (f) List: the length is unknown until the whole file is read.

Solution 5.4. **Listing 5.57** and **Listing 5.58** implement both strategies and count every copied element.

Listing 5.57 `GrowArray.java`: growing by one slot versus doubling.

```java
import java.util.Arrays;

public class GrowArray {
    static long copies = 0;                          // how many slots we have copied so far

    // Strategy 1: make room for exactly one more value every time.
    static int[] appendOne(int[] a, int value) {
        int[] b = new int[a.length + 1];
        for (int i = 0; i < a.length; i++) {
            b[i] = a[i];
            copies++;
        }
        b[a.length] = value;
        return b;
    }

    public static void main(String[] args) {
        int n = 1000;
        int[] a = new int[0];
        for (int v = 1; v <= n; v++) {
            a = appendOne(a, v);
        }
        System.out.println("grow by one:  n = " + n + ", copies = " + copies);

        // Strategy 2: double the capacity when full, and remember how much is used.
        copies = 0;
        int[] data = new int[1];
        int size = 0;
        for (int v = 1; v <= n; v++) {
            if (size == data.length) {
                copies += size;
                data = Arrays.copyOf(data, 2 * data.length);
            }
            data[size] = v;
            size++;
        }
        System.out.println("doubling:     n = " + n + ", copies = " + copies
                + ", capacity = " + data.length);
    }
}
```

```
grow by one:  n = 1000, copies = 499500
doubling:     n = 1000, copies = 1023, capacity = 1024
```

Listing 5.58 `grow_array.py`: the same experiment in Python.

```
copies = 0                                      # how many slots we have copied so far


def append_one(a, value):
    """Strategy 1: make room for exactly one more value every time."""
    global copies
    b = [0] * (len(a) + 1)
    for i in range(len(a)):
        b[i] = a[i]
        copies += 1
    b[len(a)] = value
    return b


def main():
    global copies
    n = 1000
    a = []
    for v in range(1, n + 1):
        a = append_one(a, v)
    print("grow by one:  n = " + str(n) + ", copies = " + str(copies))

    # Strategy 2: double the capacity when full, and remember how much is used.
    copies = 0
    data = [0] * 1
    size = 0
    for v in range(1, n + 1):
        if size == len(data):
            copies += size
            data = data + [0] * len(data)
        data[size] = v
        size += 1
    print("doubling:     n = " + str(n) + ", copies = " + str(copies)
          + ", capacity = " + str(len(data)))


if __name__ == "__main__":
    main()
```

```
grow by one:  n = 1000, copies = 499500
doubling:     n = 1000, copies = 1023, capacity = 1024
```

Growing by one copies $1000 \times 999/2 = 499{,}500$ elements, exactly as Equation (5.2) predicts. Doubling copies $1 + 2 + \cdots + 512 = 1023 < 2 \times 1000$ elements, as Equation (5.3) predicts, at the price of a capacity of 1024 for 1000 values. For $n = 10^6$ the first strategy would copy about 5×10^{11} elements and the second fewer than 2×10^6.

Solution 5.5. For `f(0)`: `A` is printed; `10 / 0` throws `ArithmeticException`, so `B` is skipped; the `catch` prints `C` and prepares to return -1; the `finally` prints `D`; the method returns -1. Output `A C D` on three lines. For `f(1)`: `A`, then `y = 10`, then `B`; the `return` prepares 10; `finally` prints `D`; the method returns 10. Output `A B D`. The `finally` block runs in both cases, after the value to be returned has been computed and before it is actually returned (**Figure 5.6**).

Solution 5.6. Unchecked (extend `RuntimeException`), usually programming errors: `NullPointerException`, `NumberFormatException` (though it often comes from bad input, and then must be caught), `ClassCastException`, `InputMismatchException` (again from input), `ArithmeticException`. Checked, outside problems: `FileNotFoundException`, `IOException`, and `OrderFormatException`, which extends `Exception` directly (**Figure 5.4**). Note that "unchecked" does not mean "never caught": exceptions caused by user input, like the two conversion failures, should be caught near the input.

Solution 5.7. **Listing 5.59** and **Listing 5.60**. The pair (−30,45) sums to 15 and is accepted; the rule is about the sum, not the individual values.

Listing 5.59 `SomeNumbers.java`: throwing `IllegalArgumentException` and catching it in the caller.

```
public class SomeNumbers {
    public static void getSomeNumbers(int first, int second) {
        if (first + second >= 20) {
            throw new IllegalArgumentException("The numbers you've entered are not
acceptable.");
        }
        System.out.println("Your numbers are fine. Well done.");
    }

    public static void main(String[] args) {
        int[][] pairs = {{3, 4}, {15, 5}, {-30, 45}, {9, 10}};
        for (int[] p : pairs) {
            System.out.print(p[0] + " + " + p[1] + ": ");
            try {
                getSomeNumbers(p[0], p[1]);
            } catch (IllegalArgumentException e) {
                System.out.println("refused, " + e.getMessage());
            }
        }
    }
}
```

```
3 + 4: Your numbers are fine. Well done.
15 + 5: refused, The numbers you've entered are not acceptable.
-30 + 45: Your numbers are fine. Well done.
9 + 10: Your numbers are fine. Well done.
```

Listing 5.60 `some_numbers.py`: the same with `ValueError`.

```
def get_some_numbers(first, second):
    if first + second >= 20:
        raise ValueError("The numbers you've entered are not acceptable.")
    print("Your numbers are fine. Well done.")


def main():
    pairs = [(3, 4), (15, 5), (-30, 45), (9, 10)]
    for first, second in pairs:
        print(str(first) + " + " + str(second) + ": ", end="")
        try:
            get_some_numbers(first, second)
        except ValueError as e:
            print("refused, " + str(e))


if __name__ == "__main__":
    main()
```

```
3 + 4: Your numbers are fine. Well done.
15 + 5: refused, The numbers you've entered are not acceptable.
-30 + 45: Your numbers are fine. Well done.
9 + 10: Your numbers are fine. Well done.
```

Solution 5.8. The Java version, **Listing 5.61**, uses the `hasNext`/`nextInt`/`next` pattern of **Listing 5.23**; the Python version, **Listing 5.62**, reads all the input and splits it into tokens. Both guard the division against a count of zero. The integers read are 4,8,15, −2, so the average is 25/4 = 6.25; `x`, `3.5` and `six` are skipped, and `1000` is never read because it comes after `99`.

Listing 5.61 `AverageInput.java`: robust averaging of typed integers.

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class AverageInput {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        long sum = 0;
        int count = 0;
        int skipped = 0;
        boolean sawStop = false;
        while (s.hasNext()) {
            try {
                int v = s.nextInt();
                if (v == 99) {
                    sawStop = true;
                    break;
                }
                sum += v;
                count++;
            } catch (InputMismatchException e) {
                s.next();                          // throw the bad token away
                skipped++;
            }
        }
        if (!sawStop) {
            System.out.println("(input ended without 99)");
        }
        System.out.println("numbers: " + count + ", skipped: " + skipped);
        if (count > 0) {
            System.out.printf("average: %.2f%n", (double) sum / count);
        } else {
            System.out.println("average: undefined, no numbers were read");
        }
    }
}
```

```
  4 x 8
  15 3.5 -2
  six 99 1000
```

```
numbers: 4, skipped: 3
average: 6.25
```

Listing 5.62 `average_input.py`: the same in Python.

```python
import sys


def main():
    total = 0
    count = 0
    skipped = 0
    saw_stop = False
    tokens = sys.stdin.read().split()         # every whitespace-separated token
    for token in tokens:
        try:
            v = int(token)
        except ValueError:
            skipped += 1                      # throw the bad token away
            continue
        if v == 99:
            saw_stop = True
            break
        total += v
        count += 1
    if not saw_stop:
        print("(input ended without 99)")
    print("numbers: " + str(count) + ", skipped: " + str(skipped))
    if count > 0:
        print(f"average: {total / count:.2f}")
    else:
        print("average: undefined, no numbers were read")


if __name__ == "__main__":
    main()
```

```
numbers: 4, skipped: 3
average: 6.25
```

Solution 5.9. There are three mistakes. (1) The compiler reports the catch order:

```
Buggy.java:9: error: exception NumberFormatException has already been caught
            } catch (NumberFormatException e) {
              ^
1 error
```

The general `catch (Exception e)` comes first, so the specific clause could never run; swap them. (2) The first `catch` block is empty: once the order is fixed it would still silently swallow the `ArrayIndexOutOfBoundsException` from mistake 3. (3) The loop condition `i <= prices.length` visits index 3, which does not exist; it must be `i < prices.length`. The corrected loop prints “bad price ten thousand” and “total = 12000”.

Solution 5.10. The check for insufficient balance comes *before* the subtraction, so a failed payment changes nothing. The exception is checked, so the compiler forces `main` to handle it.

Listing 5.63 `PrepaidCard.java`: a custom checked exception for an insufficient balance.

```
public class PrepaidCard {
    // A checked exception: callers must plan for a card that runs dry.
    static class InsufficientBalanceException extends Exception {
        private final long shortBy;

        InsufficientBalanceException(long balance, long amount) {
            super("balance Rp " + balance + " is short of Rp " + amount);
            this.shortBy = amount - balance;
        }

        long getShortBy() {
            return shortBy;
        }
    }

    private final String owner;
    private long balance;

    PrepaidCard(String owner, long balance) {
        this.owner = owner;
        this.balance = balance;
    }

    void pay(long amount) throws InsufficientBalanceException {
        if (amount <= 0) {
            throw new IllegalArgumentException("amount must be positive: " + amount);
        }
        if (amount > balance) {
            throw new InsufficientBalanceException(balance, amount);   // balance untouched
        }
        balance -= amount;
    }

    public static void main(String[] args) {
        PrepaidCard card = new PrepaidCard("Sari", 50000);
        long[] bills = {24500, 30000, 20000, -5};
        for (long bill : bills) {
            try {
                card.pay(bill);
                System.out.println("paid " + bill + ", balance now " + card.balance);
            } catch (InsufficientBalanceException e) {
                System.out.println("declined: " + e.getMessage()
                        + " (short by " + e.getShortBy() + ")");
            } catch (IllegalArgumentException e) {
                System.out.println("rejected: " + e.getMessage());
            }
        }
        System.out.println(card.owner + " keeps Rp " + card.balance);
    }
}
```

Listing 5.63 (Continued) *PrepaidCard.java: a custom checked exception for an insufficient balance.*

```
paid 24500, balance now 25500
declined: balance Rp 25500 is short of Rp 30000 (short by 4500)
paid 20000, balance now 5500
rejected: amount must be positive: -5
Sari keeps Rp 5500
```

Listing 5.64 `prepaid_card.py`: the same card in Python.

```
class InsufficientBalanceError(Exception):
    """Raised when a card cannot cover a payment."""

    def __init__(self, balance, amount):
        super().__init__("balance Rp " + str(balance) + " is short of Rp " + str(amount))
        self.short_by = amount - balance


class PrepaidCard:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def pay(self, amount):
        """Raises ValueError for a non-positive amount, InsufficientBalanceError if
short."""
        if amount <= 0:
            raise ValueError("amount must be positive: " + str(amount))
        if amount > self.balance:
            raise InsufficientBalanceError(self.balance, amount)   # balance untouched
        self.balance -= amount


def main():
    card = PrepaidCard("Sari", 50000)
    for bill in [24500, 30000, 20000, -5]:
        try:
            card.pay(bill)
            print("paid " + str(bill) + ", balance now " + str(card.balance))
        except InsufficientBalanceError as e:
            print("declined: " + str(e) + " (short by " + str(e.short_by) + ")")
        except ValueError as e:
            print("rejected: " + str(e))
    print(card.owner + " keeps Rp " + str(card.balance))


if __name__ == "__main__":
    main()
```

```
paid 24500, balance now 25500
declined: balance Rp 25500 is short of Rp 30000 (short by 4500)
paid 20000, balance now 5500
rejected: amount must be positive: -5
Sari keeps Rp 5500
```

Listing 5.63 and **Listing 5.64** both decline the 30,000 bill with the balance still at 25,500, then accept 20,000.

Solution 5.11. Listing 5.65 reuses the loader and keeps one counter per service in three parallel arrays indexed like `Order.SERVICES`. The multi-catch `IOException | OrderFormatException` handles both load failures with one message.

Listing 5.65 `ServiceSummary.java`: orders, kilograms and revenue per service.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.io.FileWriter;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;

public class ServiceSummary {
    public static void main(String[] args) {
        List<Order> orders;
        try {
            orders = OrderCsv.load("orders.csv", new ArrayList<>());
        } catch (IOException | OrderFormatException e) {    // one handler, two types
            System.out.println("Cannot load orders: " + e.getMessage());
            return;
        }
        String[] services = Order.SERVICES;
        int[] count = new int[services.length];              // one counter per service
        double[] kg = new double[services.length];
        long[] revenue = new long[services.length];
        for (Order o : orders) {
            for (int i = 0; i < services.length; i++) {
                if (services[i].equals(o.getService())) {
                    count[i]++;
                    kg[i] += o.getWeightKg();
                    revenue[i] += o.price();
                }
            }
        }
        try (PrintWriter out = new PrintWriter(
                new FileWriter("summary.csv", StandardCharsets.UTF_8))) {
            out.println("service,orders,kg,revenue");
            for (int i = 0; i < services.length; i++) {
                out.println(services[i] + "," + count[i] + "," + kg[i] + "," + revenue[i]);
                System.out.println(services[i] + "," + count[i] + "," + kg[i] + "," +
revenue[i]);
            }
        } catch (IOException e) {
            System.out.println("Cannot write summary: " + e.getMessage());
        }
    }
}
```

```
WASH,2,15.5,150500
IRON,1,2.0,10000
WASH_IRON,1,5.0,75000
```

Listing 5.66 `service_summary.py`: the same summary in Python.

```
import order_csv
from order import RATES
from order_errors import OrderFormatError


def main():
    try:
        orders = order_csv.load("orders.csv", [])
    except (OSError, OrderFormatError) as e:  # one handler, two types
        print("Cannot load orders: " + str(e))
        return
    services = list(RATES)                    # WASH, IRON, WASH_IRON, in that order
    count = [0] * len(services)               # one counter per service
    kg = [0.0] * len(services)
    revenue = [0] * len(services)
    for o in orders:
        i = services.index(o.service)
        count[i] += 1
        kg[i] += o.weight_kg
        revenue[i] += o.price()
    try:
        with open("summary.csv", "w", encoding="utf-8") as out:
            out.write("service,orders,kg,revenue\n")
            for i in range(len(services)):
                row = services[i] + "," + str(count[i]) + "," + str(kg[i]) + "," +
str(revenue[i])
                out.write(row + "\n")
                print(row)
    except OSError as e:
        print("Cannot write summary: " + str(e))


if __name__ == "__main__":
    main()
```

```
WASH,2,15.5,150500
IRON,1,2.0,10000
WASH_IRON,1,5.0,75000
```

The four valid orders of `orders.csv` give `WASH` 24,500 + 126,000 = 150,500, `IRON` 10,000 and `WASH_IRON` 75,000. The same lines are written to `summary.csv`. In **Listing 5.66**, `list(RATES)` gives the dictionary's keys in insertion order.

Solution 5.12. A `Scanner` does not read one token at a time from the operating system; it reads a large block of input into its own buffer and then hands out tokens from that buffer. When the input comes from a file, the first `Scanner` has probably already read the *whole* file into its buffer. A new `Scanner` built on the same `System.in` starts reading where the operating system left off, which is the end of the file, so it finds nothing and its first `nextInt()` throws `NoSuchElementException`; everything still in the old buffer is lost. At the keyboard, input arrives one line at a time, so the old buffer usually holds only the bad line and the mistake goes unnoticed. The robust fix is to keep one `Scanner` and discard the bad token with `next()`.

Solution 5.13. **Listing 5.67** opens the file with `try`-with-resources and reports the two failure cases separately. `split("\\s+")` splits at runs of whitespace; the line is trimmed first, and an empty line counts no words. Python's `split()` with no argument does all of this by itself (**Listing 5.68**).

Listing 5.67 `WordCount.java`: counting lines, words and characters of a file named on the command line.

```java
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.nio.charset.StandardCharsets;

public class WordCount {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("usage: java WordCount <file>");
            return;
        }
        int lines = 0;
        int words = 0;
        int chars = 0;
        try (BufferedReader in = new BufferedReader(
                new FileReader(args[0], StandardCharsets.UTF_8))) {
            String line;
            while ((line = in.readLine()) != null) {
                lines++;
                chars += line.length();
                String trimmed = line.trim();
                if (!trimmed.isEmpty()) {
                    words += trimmed.split("\\s+").length;   // one or more spaces
                }
            }
        } catch (FileNotFoundException e) {
            System.out.println("no such file: " + args[0]);
            return;
        } catch (IOException e) {
            System.out.println("cannot read " + args[0] + ": " + e.getMessage());
            return;
        }
        System.out.println(lines + " lines, " + words + " words, "
                + chars + " characters (line breaks not counted) in " + args[0]);
    }
}
```

Running `java WordCount notes.txt`, then `java WordCount nope.txt`, then `java WordCount` gives:

```
3 lines, 17 words, 99 characters (line breaks not counted) in notes.txt
no such file: nope.txt
usage: java WordCount <file>
```

Listing 5.68 `word_count.py`: the same counts in Python.

```
import sys


def main():
    if len(sys.argv) != 2:
        print("usage: python word_count.py <file>")
        return
    name = sys.argv[1]
    lines = words = chars = 0
    try:
        with open(name, encoding="utf-8") as f:
            for line in f:
                line = line.rstrip("\n")
                lines += 1
                chars += len(line)
                words += len(line.split())    # split() with no argument: any run of spaces
    except FileNotFoundError:
        print("no such file: " + name)
        return
    except OSError as e:
        print("cannot read " + name + ": " + str(e))
        return
    print(str(lines) + " lines, " + str(words) + " words, " + str(chars)
          + " characters (line breaks not counted) in " + name)


if __name__ == "__main__":
    main()
```

```
3 lines, 17 words, 99 characters (line breaks not counted) in notes.txt
no such file: nope.txt
usage: python word_count.py <file>
```

Further reading

The Java Tutorials [6], [7], [8] cover arrays, exceptions and basic I/O step by step, and the API documentation of `ArrayList` [9] and `Scanner` [10] is worth reading in full once. The precise rules for checked exceptions are in Chapter 11 of the Java Language Specification [3]. Horstmann [11] gives a thorough textbook treatment of all of this chapter's Java. For Python, the official tutorial's chapters on data structures [12] and on errors and exceptions [13], the list of built-in exceptions [1], and the `csv` module documentation [5] are the primary sources. Bloch [2] collects the accepted wisdom on using exceptions well, and RFC 4180 [4] describes the CSV format.

References

[1] Python Software Foundation, "Built-in Exceptions," *The Python Standard Library*. [Online]. Available: https://docs.python.org/3/library/exceptions.html

[2] J. Bloch, *Effective Java*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2018, ch. 10, "Exceptions."

[3] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*, ch. 11, "Exceptions." Oracle, 2023.

[4] Y. Shafranovich, "Common Format and MIME Type for Comma-Separated Values (CSV) Files," IETF RFC 4180, Oct. 2005.

[5] Python Software Foundation, "csv — CSV File Reading and Writing," *The Python Standard Library*. [Online]. Available: https://docs.python.org/3/library/csv.html

[6] Oracle, "Arrays," *The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

[7] Oracle, "Lesson: Exceptions," *The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions/

[8] Oracle, "Lesson: Basic I/O" and "The try-with-resources Statement," *The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/io/

[9] Oracle, "Class ArrayList," *Java SE 21 & JDK 21 API Specification*, 2023. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html

[10] Oracle, "Class Scanner," *Java SE 21 & JDK 21 API Specification*, 2023. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Scanner.html

[11] C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed. Hoboken, NJ, USA: Pearson, 2022.

[12] Python Software Foundation, "Data Structures," *The Python Tutorial*, ch. 5. [Online]. Available: https://docs.python.org/3/tutorial/datastructures.html

[13] Python Software Foundation, "Errors and Exceptions," *The Python Tutorial*, ch. 8. [Online]. Available: https://docs.python.org/3/tutorial/errors.html

Chapter 6

Documentation, Testing, and Interfaces

By the end of this chapter you will be able to document your classes with Javadoc and Python docstrings and turn the comments into reference pages with `javadoc` and `pydoc`; test a class from zero, first with a plain program that checks itself and then with JUnit 6, `unittest` and pytest, using arrange–act–assert, edge cases and tests written before the code; make `equals`/`hashCode` and `__eq__`/`__hash__` agree so that sets and maps work; give objects a readable printed form; overload methods in Java and explain why Python does the same job differently; and, most importantly, define an *interface* — a promise of what an object can do, with no word about how — and write code that depends only on that promise. The chapter ends with a pricing engine for Profitina Laundry in which five price rules sit behind one interface, every rule fully tested in both languages.

6.1 Where We Left Off

Chapter 5 taught our programs to hold many things at once, to read input, to survive bad data with exceptions, and to keep results in files. Real enough to run a small shop for a morning — but not yet trustworthy. Ask three awkward questions about any class we have written so far.

How does a colleague know how to use it? Today she must read the source, which works for twenty lines and not for the thousands of classes of the Java library. *How do we know it works?* We ran it once and nodded; after the next change nobody will look again. *How do we change it without breaking everything that uses it?* If fifty places call a pricing method directly, changing prices means visiting fifty places.

This chapter answers the three questions in order. **Documentation** — comments in a fixed format that tools turn into reference pages — answers the first. **Testing** — small programs that compare what your code does with what it should do, every time, in a second — answers the second. **Interfaces** — a written-down promise of *what* a group of objects can do — answer the third, because the rest of the program can then depend on the promise and never on the details behind it.

Along the way we meet three smaller tools the big three rely on: several constructors for one class, the methods that decide whether two objects are equal, and several versions of one method name. Each hides a trap, and we will walk into each one on purpose, with real output, so that you never walk into it by accident.

> ✓ **KEY INSIGHT.** Documentation, tests and interfaces are three ways of writing down a *promise*. A doc comment states the promise in words for people. A test states it in code and checks it. An interface states it in the type system, so that the compiler (Java) or a type checker (Python) checks it. Good software keeps all three in step.

6.2 A Class, Looked at Once More

Before we document or test a class, let us agree on its parts. A typical class is laid out from top to bottom like this.

At the top are the **fields**: the variables that belong to each object, which every method of the class can see and change. We mark them `private` in Java, or start their names with an underscore in Python, so that only the class itself touches them.

Next come the **constructors**, which run when the program writes `new BankAccount(...)` in Java or `BankAccount(...)` in Python and give every field a sensible first value. A Java class that declares no constructor gets an empty one with no parameters from the compiler.

Then come the **methods**: usually the *getters* first, then the *setters*, and at the bottom the methods every class inherits and usually improves — `toString`, which says how the object looks when printed, and `equals`, which says when two objects count as the same (Sections 6.9 and 6.10).

The order is a convention, not a rule; both languages accept the members in any order.

6.2.1 More than one way to be born: several constructors

A bank lets a customer open an account in two ways. She can open it with money, in which case the bank grants an overdraft of 50 (she may go 50 below zero). Or she can open it empty, and then the overdraft is only 10. The account is the same kind of thing either way; only the *starting values* differ, and they differ according to what the customer brought.

Java expresses this with **two constructors** in the same class. They share the class name, so they must differ in their parameter lists: one takes a `double`, the other takes nothing. When the program says `new BankAccount()` the compiler picks the constructor with no parameters; when it says `new BankAccount(25)` it picks the one with a `double`. **Listing 6.1** shows the whole class. Look past the long comments for a moment — they are the subject of Section 6.3 — and find the two constructors.

Listing 6.1 `BankAccount.java`: a class with two constructors and a documentation comment on every member.

```java
/**
 * A bank account with a balance and an overdraft limit.
 *
 * <p>An account opened with money gets an overdraft of 50;
 * an account opened empty gets an overdraft of only 10.</p>
 */
public class BankAccount {
    private double balance;
    private int overdraft;

    /**
     * Opens an account with an opening deposit.
     *
     * @param openingAmount the money paid in on the first day
     */
    public BankAccount(double openingAmount) {
        balance = openingAmount;
        overdraft = 50;
    }

    /**
     * Opens an empty account. No amount is supplied,
     * so the smaller default overdraft applies.
     */
    public BankAccount() {
        balance = 0;
        overdraft = 10;
    }

    /**
     * Returns the current balance.
     *
     * @return the balance
     */
    public double getBalance() {
        return balance;
    }
```

Listing 6.1 (Continued) *BankAccount.java: a class with two constructors and a documentation comment on every member.*

```
    /**
     * Returns how far the balance may go below zero.
     *
     * @return the overdraft limit
     */
    public int getOverdraft() {
        return overdraft;
    }

    /**
     * Calculates the compound interest earned on an amount.
     *
     * @param initial the initial balance
     * @param years   the number of years
     * @param rate    the yearly interest rate, for example 0.06 for 6%
     * @return the compound interest on the balance
     * @throws IllegalArgumentException if {@code years} is negative
     */
    public double interestOn(double initial, int years, double rate) {
        if (years < 0) {
            throw new IllegalArgumentException("years must not be negative");
        }
        double total = initial * Math.pow(1 + rate, years);
        return total - initial;
    }

    /**
     * Describes the account for printing.
     *
     * @return the balance and the overdraft in one line
     */
    @Override
    public String toString() {
        return "BankAccount[balance=" + balance + ", overdraft=" + overdraft + "]";
    }
```

Listing 6.1 (Continued) *BankAccount.java: a class with two constructors and a documentation comment on every member.*

```
    /**
     * Opens two accounts and prints them.
     *
     * @param args not used
     */
    public static void main(String[] args) {
        BankAccount ba1 = new BankAccount();      // the no-argument constructor
        BankAccount ba2 = new BankAccount(25);    // the one-argument constructor
        System.out.println(ba1);
        System.out.println(ba2);
        double interest = ba2.interestOn(1_000_000, 2, 0.06);
        System.out.println("Interest on 1,000,000 at 6% for 2 years: " +
Math.round(interest));
    }
}
```

Compiled with `javac BankAccount.java` and run with `java BankAccount`, it prints:

```
BankAccount[balance=0.0, overdraft=10]
BankAccount[balance=25.0, overdraft=50]
Interest on 1,000,000 at 6% for 2 years: 123600
```

The first account, `ba1`, was built by the constructor with no parameters, so its balance is `0.0` and its overdraft `10`. The second, `ba2`, was built by the one-parameter constructor: balance `25.0`, overdraft `50`. The third line calls `interestOn`, which we explain in Section 6.3.

Python allows only **one** `__init__` per class. If you write a second one, it simply replaces the first (Section 6.11 shows this very behavior with ordinary methods). Python reaches the same goal with a **default argument**: a parameter that has a value written into the function header, used whenever the caller leaves it out. **Listing 6.2** gives `opening_amount` the default `None`, the Python value that means "nothing here", and then asks which case it is in.

Listing 6.2 `bank_account.py`: one `__init__` with a default argument does the work of Java's two constructors.

```
"""A bank account with a balance and an overdraft limit.

Run this file to open two accounts, or run
``python -m doctest -v bank_account.py`` to check the examples
written in the docstrings.
"""


class BankAccount:
    """A bank account with a balance and an overdraft limit.

    An account opened with money gets an overdraft of 50;
    an account opened empty gets an overdraft of only 10.

    >>> BankAccount().overdraft
    10
    >>> BankAccount(25).overdraft
    50
    """

    def __init__(self, opening_amount=None):
        """Open an account, with or without an opening deposit.

        Args:
            opening_amount: the money paid in on the first day, or
                None (the default) to open an empty account.
        """
        if opening_amount is None:          # no amount supplied: the empty account
            self._balance = 0.0
            self._overdraft = 10
        else:
            self._balance = float(opening_amount)
            self._overdraft = 50

    @property
    def balance(self):
        """The current balance."""
        return self._balance

    @property
    def overdraft(self):
        """How far the balance may go below zero."""
        return self._overdraft
```

Listing 6.2 (Continued) *bank_account.py: one __init__ with a default argument does the work of Java's two constructors.*

```
    def interest_on(self, initial, years, rate):
        """Calculate the compound interest earned on an amount.

        Args:
            initial: the initial balance.
            years: the number of years.
            rate: the yearly interest rate, for example 0.06 for 6%.

        Returns:
            The compound interest on the balance.

        Raises:
            ValueError: if years is negative.

        >>> round(BankAccount().interest_on(1_000_000, 2, 0.06))
        123600
        """
        if years < 0:
            raise ValueError("years must not be negative")
        total = initial * (1 + rate) ** years
        return total - initial

    def __repr__(self):
        return f"BankAccount[balance={self._balance}, overdraft={self._overdraft}]"


if __name__ == "__main__":
    ba1 = BankAccount()        # no argument: the default None is used
    ba2 = BankAccount(25)
    print(ba1)
    print(ba2)
    interest = ba2.interest_on(1_000_000, 2, 0.06)
    print("Interest on 1,000,000 at 6% for 2 years:", round(interest))
```

Run with `python3 bank_account.py`, it prints exactly the same three lines as the Java program:

```
BankAccount[balance=0.0, overdraft=10]
BankAccount[balance=25.0, overdraft=50]
Interest on 1,000,000 at 6% for 2 years: 123600
```

Why `None` and not `0`? Because "opened with zero money" and "opened with nothing" are different situations to this bank: `BankAccount(0)` is a deposit that happens to be zero and earns the larger overdraft, exactly like `new BankAccount(0)` in Java. The `@property` lines make `balance` and `overdraft` read-only attributes, as a private field with a public getter does in Java.

> ⚠ **THINK.** In Java the two constructors repeat the job of setting two fields. With two fields that is harmless; with ten it invites the day when one constructor sets nine and forgets the tenth. Java lets one constructor call another with `this(...)` as its very first statement, so that all the real work happens in one place. Exercise 6.3 asks you to rewrite `BankAccount` that way, and Python's answer — alternative constructors written as class methods — appears in its solution.

6.3 Documentation Comments: Javadoc and Docstrings

You already write comments. A comment such as `// the no-argument constructor` in **Listing 6.1** speaks to somebody *reading the code*: it explains a line.

There is a second audience that never reads your code at all. A colleague who wants to *use* `BankAccount` needs to know what each public method does, what it expects, what it returns and what can go wrong — and nothing about how it is done inside. That is exactly what the Java API pages tell you about `String` or `ArrayList`: you have used those classes for five chapters without ever seeing their source.

Comments for that second audience are called **documentation comments**, and both languages have a fixed format for them so that a tool can collect them into reference pages.

> **DEFINITION.** A *documentation comment* describes the *interface* of a class or method — its purpose, its parameters, its result and its failures — in a format that a documentation tool can extract. In Java it is a *Javadoc comment*, which begins with `/**` and ends with `*/` and is placed immediately before the class, field, constructor or method it describes. In Python it is a *docstring*: a string literal written as the very first statement inside the module, class or function it describes.

6.3.1 Javadoc, step by step

Take the simplest possible method, a getter:

```
public class Pet {
    private final String name = "Rover";

    /**
     * Gets the pet's name.
     *
     * @return the pet's name
     */
    public String getName() {
        return name;
    }
}
```

The comment opens with a slash and **two** asterisks, `/**`, and closes with `*/`. The asterisks at the start of the lines in between are decoration; the tool strips them. The first sentence, “Gets the pet’s name.”, is the *summary*: it appears in the method table at the top of the generated page, so it should say in a few words what the method does. After a blank line come **block tags**, words beginning with `@` that label a particular piece of information. `@return` describes the value the method gives back.

A method with parameters describes each of them, in order, with one `@param` tag each, in the form `@param <parameterName> <description>`. The name must be spelled exactly as it is in the code — the tool checks. The method `interestOn` in **Listing 6.1** has three parameters, a result and a failure, so its comment carries three `@param` tags, one `@return` and one `@throws`:

```
/**
 * Calculates the compound interest earned on an amount.
 *
 * @param initial the initial balance
 * @param years   the number of years
 * @param rate    the yearly interest rate, for example 0.06 for 6%
 * @return the compound interest on the balance
 * @throws IllegalArgumentException if {@code years} is negative
 */
```

The method computes *compound* interest: each year the interest is added to the balance, and the next year’s interest is earned on the larger amount. After one year an initial amount P at yearly rate r has grown to $P(1+r)$; after two years to $P(1+r)(1+r) = P(1+r)^2$; after n years to $P(1+r)^n$. The interest is what was added to the original amount:

$$I = P(1 + r)^n - P \tag{6.1}$$

Equation (6.1) is exactly the two lines in the body of `interestOn`: `total = initial * Math.pow(1 + rate, years)` computes $P(1 + r)^n$, and `total - initial` subtracts P. With $P = 1\,000\,000$, $r = 0.06$ and $n = 2$ we get $1\,000\,000 \times 1.06^2 = 1\,000\,000 \times 1.1236 = 1\,123\,600$, so $I = 123\,600$ — the number both programs printed.

The inline tag `{@code years}` inside the `@throws` line tells the tool to print `years` in code font. Inline tags live *inside* a sentence and are wrapped in braces; block tags start a line. **Table 6.1** lists the tags you will use most, with the equivalent section of a Python docstring.

Table 6.1 The Javadoc tags used most often, and where the same information goes in a Python docstring.

Javadoc tag	What it documents	Python docstring (Google style)
first sentence	a one-line summary	the first line
`@param name text`	one parameter, in order	an entry under `Args:`
`@return text`	the value returned (omit for `void`)	`Returns:`
`@throws Type text`	an exception and when it happens	`Raises:`
`{@code text}`	code font inside a sentence	backquotes, `` `text` ``
`{@link Class#method}`	a clickable link to another item	the name, e.g. `Dog.age`
`@see item`	"see also" at the end of the page	a `See Also:` section
`@since version`	the version that added the item	a note in the text
`@deprecated text`	"do not use any more; use … instead"	a `.. deprecated::` note

Everything a caller needs belongs in the comment; nothing about the inside of the method does. "Uses `Math.pow`" is an implementation detail. "The rate is a fraction, so 6% is written 0.06" is part of the contract, because a caller who passes `6` would otherwise receive a very generous bank.

> ⚠ **TRAP.** A Javadoc comment must end with `*/`. Suppose that, typing quickly, you end the `interestOn` comment with a lonely `/` instead. The compiler does not complain — it treats everything up to the *next* `*/` it finds as part of the comment. If the next method has a Javadoc comment, the whole of `interestOn`, header and body, disappears into the comment. We tried it with a class that had `interestOn` followed by a documented method `twice`. `javac` compiled it without a word; asking the compiled class what it contains (`javap Interest`) gave

```
Compiled from "Interest.java"
public class Interest {
  public Interest();
  public double twice(double);
}
```

> and a program that called the method was then refused with

```
UseInterest.java:3: error: cannot find symbol
        System.out.println(new Interest().interestOn(1000, 2, 0.06));
                                         ^
  symbol:   method interestOn(int,int,double)
  location: class Interest
1 error
```

> The method is missing because you commented it out. An editor with syntax coloring shows the whole method in comment color — one more reason to use one.

6.3.2 Docstrings, step by step

Python's docstrings follow the conventions of PEP 257 [1]. A docstring is written with three double quotes, `"""`, so that it can span several lines. Its first line is a one-sentence summary ending with a period, written as a command ("Open an account", "Return the balance"). If there is more to say, a blank line follows, then the details.

PEP 257 fixes only the outline; the inside of a long docstring follows one of several popular styles. This book uses the *Google style*, which you can see in **Listing 6.2**: headed sections `Args:`, `Returns:` and `Raises:`, each followed by indented entries. It reads well in plain text, and every Python documentation tool understands it.

Python keeps every docstring at run time. Type `help(BankAccount.interest_on)` at the Python prompt and the docstring is printed; the attribute `BankAccount.interest_on.__doc__` holds the same text as an ordinary string. Java comments, by contrast, are thrown away by the compiler; only the `javadoc` tool reads them.

6.3.3 Examples that check themselves: doctest

Look again at the class docstring in **Listing 6.2**. It contains lines that look like a Python session:

```
>>> BankAccount().overdraft
10
```

The standard module `doctest` [2] finds every such pair — a line starting with `>>>` and the output that should follow it — runs the code, and compares. The command `python3 -m doctest -v bank_account.py` runs all three examples in the file; its last lines read

```
   2 tests in bank_account.BankAccount
   1 tests in bank_account.BankAccount.interest_on
3 tests in 7 items.
3 passed and 0 failed.
Test passed.
```

A doctest is a documentation example that can never go out of date without somebody noticing. It is also our first glimpse of testing: code that runs other code and compares the result with an expectation. Java has no direct equivalent; since Java 18 a `{@snippet}` tag lets you include checked example code in Javadoc, but the proper home for Java checks is a test class, which Section 6.7 builds.

6.4 Generating and Reading Documentation

A documentation comment pays off when a tool turns it into pages. **Figure 6.1** shows the three routes this chapter uses: `javadoc` for Java, `pydoc` for Python, and `doctest`, which reads the same docstrings but *runs* their examples instead of printing them.

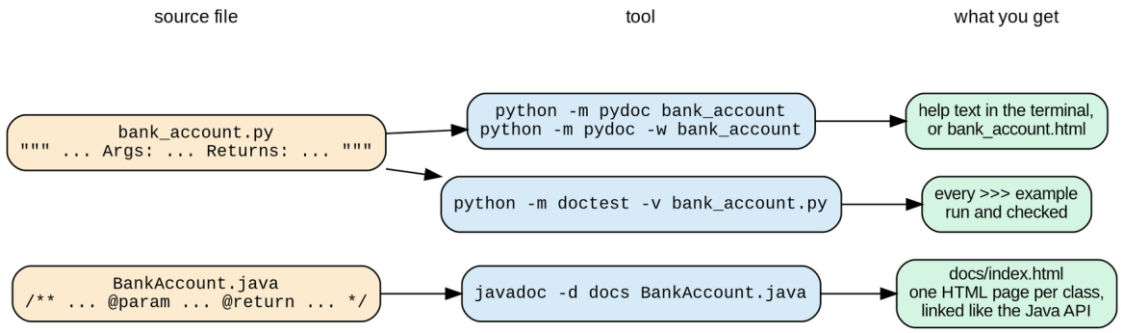


Figure 6.1 From documentation comments to documentation: `javadoc` builds linked HTML pages, `pydoc` prints help text or writes an HTML page, and `doctest` runs the examples written in docstrings.

6.4.1 javadoc

`javadoc` is installed with every JDK, next to `javac` and `java`, and its reference manual [3] lists every option and tag. In a terminal opened in the folder that holds `BankAccount.java`, type

```
javadoc -d docs BankAccount.java
```

The option `-d docs` names the folder that receives the pages. (To document every class in the folder, write `javadoc -d docs *.java`.) The tool reports what it does:

```
Loading source file BankAccount.java...
Constructing Javadoc information...
Creating destination directory: "docs/"
Building index for all the packages and classes...
Standard Doclet version 21.0.10+7-Ubuntu-124.04
Building tree for all the packages and classes...
Generating docs/BankAccount.html...
Generating docs/package-summary.html...
Generating docs/package-tree.html...
Generating docs/overview-tree.html...
Building index for all classes...
Generating docs/allclasses-index.html...
Generating docs/allpackages-index.html...
Generating docs/index-all.html...
Generating docs/search.html...
Generating docs/index.html...
Generating docs/help-doc.html...
```

Open `docs/index.html` in a browser and you will find a page that looks just like the official Java API: the class description at the top, a *Constructor Summary* with both constructors, a *Method Summary* table whose right-hand column holds each summary sentence, and below it one detailed entry per method with its parameters, return value and exceptions. (The version line will show your own JDK.)

`javadoc` also *checks* the comments. Here is a method whose comment misspells one parameter and forgets another:

```java
/** Interest helpers. */
public class Interest2 {
    /** Creates the helper. */
    public Interest2() {
    }

    /**
     * Calculates interest after a certain number of years.
     *
     * @param initial the initial balance
     * @param year the number of years
     * @return the compound interest on the balance
     */
    public double interestOn(double initial, int years, double rate) {
        double total = initial * Math.pow(1 + rate, years);
        return total - initial;
    }
}
```

and here is what `javadoc -quiet -d out Interest2.java` says about it:

```
Interest2.java:11: error: @param name not found
     * @param year the number of years
             ^
Interest2.java:14: warning: no @param for years
    public double interestOn(double initial, int years, double rate) {
                  ^
Interest2.java:14: warning: no @param for rate
    public double interestOn(double initial, int years, double rate) {
                  ^
1 error
2 warnings
```

That is a documentation *test*, for free. Make it a habit: document every public class, constructor and method, run `javadoc`, and fix every error and warning until it runs silently. `BankAccount` was written that way, which is why its run above shows no warnings.

6.4.2 pydoc

Python's `pydoc` module reads docstrings. `python3 -m pydoc bank_account` prints the documentation of the module in the terminal (it is what `help()` shows), and `python3 -m pydoc -w bank_account` writes it to `bank_account.html`. The terminal version for **Listing 6.2** begins like this (the complete text is about sixty lines):

```
Help on module bank_account:

NAME
    bank_account - A bank account with a balance and an overdraft limit.

DESCRIPTION
    Run this file to open two accounts, or run
    ``python -m doctest -v bank_account.py`` to check the examples
    written in the docstrings.

CLASSES
    builtins.object
        BankAccount

    class BankAccount(builtins.object)
     |  BankAccount(opening_amount=None)
     |
     |  A bank account with a balance and an overdraft limit.
```

Further down, `pydoc` lists `interest_on` with its `Args:`, `Returns:` and `Raises:` sections exactly as written, and it lists `balance` and `overdraft` under "Readonly properties" — it noticed the `@property` decorators by itself. Larger Python projects use Sphinx, which builds a whole website from the same docstrings; the docstrings you write today need no change to work there.

6.4.3 Reading the API: the class `Random`

Generating documentation is half the skill; the other half is *reading* it. The official Java API documentation is exactly what `javadoc` produced for us, only for thousands of classes.

Suppose a laundry program must decide at random which of four washing machines takes the next load (a simple way to spread the wear). Java offers `Math.random()`, which returns a `double` between 0 (inclusive) and 1 (exclusive), and the class `java.util.Random`, whose page in the API lists methods for many kinds of randomness: `nextDouble()`, `nextInt()`, `nextInt(bound)`, `nextBoolean()` and more. Its summary table tells you that `nextInt(10)` returns a value from 0 (inclusive) to 10 (exclusive) — so a machine number from 1 to 4 is `nextInt(4) + 1`. **Listing 6.3** uses these methods.

Listing 6.3 `RandomDemo.java`: the main methods of `java.util.Random`, with a fixed seed.

```
import java.util.Random;

/**
 * Shows the main methods of {@code java.util.Random}.
 */
public class RandomDemo {
    /**
     * Draws a few random values and assigns eight loads to four machines.
     *
     * @param args not used
     */
    public static void main(String[] args) {
        Random rand = new Random(2026);   // a fixed seed: the same numbers every run
        double randomNumber = rand.nextDouble();      // 0.0 (inclusive) to 1.0 (exclusive)
        int randomInt = rand.nextInt();               // any int at all
        int upperBoundedInt = rand.nextInt(10);       // 0 (inclusive) to 10 (exclusive)
        boolean coin = rand.nextBoolean();
        System.out.println("nextDouble() = " + randomNumber);
        System.out.println("nextInt()    = " + randomInt);
        System.out.println("nextInt(10)  = " + upperBoundedInt);
        System.out.println("nextBoolean()= " + coin);

        System.out.print("Machines for 8 loads:");
        for (int load = 1; load <= 8; load++) {
            int machine = rand.nextInt(4) + 1;        // 1, 2, 3 or 4
            System.out.print(" " + machine);
        }
        System.out.println();
    }
}
```

```
nextDouble() = 0.6186327814734252
nextInt()    = -647640901
nextInt(10)  = 0
nextBoolean()= true
Machines for 8 loads: 1 3 3 4 1 1 4 4
```

Python's documentation for the module `random` offers `random()`, `randrange(stop)`, `randint(a, b)` (which, unlike `randrange`, *includes* its upper end *b*), `choice(sequence)` and `getrandbits(k)`. **Listing 6.4** is the same program.

Listing 6.4 `random_demo.py`: the same ideas with Python's `random` module.

```
"""Shows the main functions of Python's random module."""
import random

rand = random.Random(2026)          # a fixed seed: the same numbers every run
random_number = rand.random()       # 0.0 (inclusive) to 1.0 (exclusive)
any_int = rand.getrandbits(32)      # 32 random bits, like Java's nextInt()
upper_bounded_int = rand.randrange(10)   # 0 (inclusive) to 10 (exclusive)
coin = rand.choice([True, False])
print("random()        =", random_number)
print("getrandbits(32) =", any_int)
print("randrange(10)   =", upper_bounded_int)
print("choice(...)     =", coin)

machines = [rand.randint(1, 4) for load in range(8)]   # randint includes both ends
print("Machines for 8 loads:", *machines)
```

```
random()        = 0.11911988496396309
getrandbits(32) = 2158288730
randrange(10)   = 8
choice(...)     = True
Machines for 8 loads: 2 4 4 4 2 1 1 1
```

The numbers differ between the two languages — each uses its own generator — but each program prints the *same* numbers every time you run it, on every computer. That is the work of the **seed**, the `2026` passed when the generator is created. A generator started from the same seed produces the same sequence. Without a seed (`new Random()`, `random.Random()`), the sequence changes on every run. For a real laundry you want the second; for a *test* you want the first, because a test whose input changes on every run cannot tell you whether a new result is a bug or a coincidence. Hold on to that thought: it is the first rule of the next section.

6.5 Testing from Zero

A program is *correct* when it does what its documentation promises for every input it accepts. We cannot try every input — `interestOn` accepts more combinations of doubles than there are atoms in a laundry — so we do the next best thing: we choose a handful of inputs cleverly, work out by hand what the answer must be, and let the computer compare. That is a **test**.

> **DEFINITION.** A *unit test* is a small piece of code that runs one small part of a program (a *unit*: usually one method of one class) on a chosen input, compares the actual result with the *expected* result worked out beforehand, and reports *pass* if they agree and *fail* if they do not. A *test suite* is a collection of unit tests that can be run together with one command.

Why bother, when you can run the program and look? Three reasons, in increasing order of importance.

1. **Looking is unreliable.** A human reading forty lines of output will miss that line 23 says `27000` instead of `28000`; a comparison in code never misses.
2. **Looking is not repeated.** A test suite runs again in a second, after every change, and says exactly which promise was broken.
3. **Tests change how you design.** A class that is hard to test — because it reads the keyboard in the middle of a calculation or picks random numbers without a seed — is almost always badly divided. Tests push you toward small methods with clear inputs and outputs, and interfaces, as the end of the chapter shows, make testing easy for exactly this reason.

6.5.1 The class under test

We start with a class small enough to hold in one hand. A `Dog` has a name and an age; a dog's age goes up by one on its birthday; a negative age is nonsense and must be refused. **Listing 6.5** and **Listing 6.6** are the two versions, documented as Section 6.3 taught.

Listing 6.5 `Dog.java`: the class we are going to test. It has no `main`; the test programs below use it.

```
/**
 * A dog with a name and an age in whole years.
 */
public class Dog {
    private final String name;
    private int age;

    /**
     * Creates a dog.
     *
     * @param name the dog's name
     * @param age  the dog's age in years
     * @throws IllegalArgumentException if {@code age} is negative
     */
    public Dog(String name, int age) {
        if (age < 0) {
            throw new IllegalArgumentException("age must not be negative: " + age);
        }
        this.name = name;
        this.age = age;
    }

    /**
     * Gets the dog's name.
     *
     * @return the dog's name
     */
    public String getName() {
        return name;
    }

    /**
     * Gets the dog's age.
     *
     * @return the age in years
     */
    public int getAge() {
        return age;
    }
```

Listing 6.5 (Continued) *Dog.java: the class we are going to test. It has no main; the test programs below use it.*

```
    /**
     * Makes the dog one year older (call it on the dog's birthday).
     */
    public void incrementAge() {
        age++;
    }
}
```

Listing 6.6 `dog.py`: the same class in Python, with read-only properties.

```
"""A dog with a name and an age in whole years."""


class Dog:
    """A dog with a name and an age in whole years."""

    def __init__(self, name, age):
        """Create a dog.

        Args:
            name: the dog's name.
            age: the dog's age in years.

        Raises:
            ValueError: if age is negative.
        """
        if age < 0:
            raise ValueError(f"age must not be negative: {age}")
        self._name = name
        self._age = age

    @property
    def name(self):
        """The dog's name."""
        return self._name

    @property
    def age(self):
        """The dog's age in years."""
        return self._age

    def increment_age(self):
        """Make the dog one year older (call it on the dog's birthday)."""
        self._age += 1
```

Neither file prints anything on its own. That is normal for a *library class*: its job is to be used by other code, and the first other code to use it will be its tests.

6.5.2 A test with no library at all

You do not need any testing framework to test. All a test needs is a way to say "I expected this, I got that, do they agree?" **Listing 6.7** writes that as a method of five lines, `check`, and then uses it four times.

Listing 6.7 `DogCheck.java`: testing `Dog` with a hand-written `check` method and no test library.

```
/**
 * Tests {@link Dog} without any test library: every check prints PASS or FAIL.
 */
public class DogCheck {
    private static int passed = 0;
    private static int failed = 0;

    /**
     * Compares what we expected with what actually happened, and reports it.
     *
     * @param what     a short description of the check
     * @param expected the value a correct Dog must produce
     * @param actual   the value this Dog did produce
     */
    static void check(String what, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS  " + what);
        } else {
            failed++;
            System.out.println("FAIL  " + what + ": expected " + expected + " but was " +
actual);
        }
    }

    /**
     * Runs every check, then prints a one-line verdict.
     *
     * @param args not used
     */
    public static void main(String[] args) {
        // Arrange: build the object under test
        Dog rover = new Dog("Rover", 5);
        // Act and assert, one behavior at a time
        check("age is 5", 5, rover.getAge());
        rover.incrementAge();
        check("age is now 6", 6, rover.getAge());
        check("name is Rover", "Rover", rover.getName());

        // An edge case: a negative age must be refused
        try {
            new Dog("Rex", -1);
            check("negative age is rejected", "exception", "no exception");
        } catch (IllegalArgumentException e) {
            check("negative age is rejected", "exception", "exception");
        }
```

Listing 6.7 (Continued) *DogCheck.java: testing Dog with a hand-written check method and no test library.*

```
        System.out.println(passed + " passed, " + failed + " failed");
        if (failed > 0) {
            System.exit(1);   // a non-zero exit code tells scripts that something broke
        }
    }
}
```

Compile both classes and run the checker (`javac Dog.java DogCheck.java`, then `java DogCheck`):

```
PASS  age is 5
PASS  age is now 6
PASS  name is Rover
PASS  negative age is rejected
4 passed, 0 failed
```

Read `main` from top to bottom and you will see the pattern every good test follows: *build* what we need (Rover, aged 5), *do* something (ask the age, add a year), and *check* the result against a number worked out *before* running anything. The last check is an **edge case** — a value at the border of what the class accepts — and it tests a *failure*: it passes only when `new Dog("Rex", -1)` throws. A suite that tries only the pleasant cases leaves the most dangerous code unexplored.

The final lines matter too: after printing its verdict, the program *exits with a non-zero status* if anything failed, because build tools, editors and servers that check every change read that status.

Listing 6.8 is the same checker in Python.

Listing 6.8 `dog_check.py`: the same self-checking program in Python.

```python
"""Test Dog without any test library: every check prints PASS or FAIL."""
import sys

from dog import Dog

passed = 0
failed = 0


def check(what, expected, actual):
    """Compare what we expected with what actually happened, and report it."""
    global passed, failed
    if expected == actual:
        passed += 1
        print("PASS ", what)
    else:
        failed += 1
        print("FAIL ", what + f": expected {expected!r} but was {actual!r}")


# Arrange: build the object under test
rover = Dog("Rover", 5)
# Act and assert, one behavior at a time
check("age is 5", 5, rover.age)
rover.increment_age()
check("age is now 6", 6, rover.age)
check("name is Rover", "Rover", rover.name)

# An edge case: a negative age must be refused
try:
    Dog("Rex", -1)
    check("negative age is rejected", "exception", "no exception")
except ValueError:
    check("negative age is rejected", "exception", "exception")

print(f"{passed} passed, {failed} failed")
if failed > 0:
    sys.exit(1)       # a non-zero exit code tells scripts that something broke
```

```
PASS  age is 5
PASS  age is now 6
PASS  name is Rover
PASS  negative age is rejected
4 passed, 0 failed
```

Now let us break the dog on purpose. Change `age++` to `age += 2` in `Dog.java` (or `self._age += 1` to `self._age += 2` in `dog.py`), save, and run the checker again. Both languages report the same thing:

```
PASS  age is 5
FAIL  age is now 6: expected 6 but was 7
PASS  name is Rover
PASS  negative age is rejected
3 passed, 1 failed
```

and the exit status is 1. The message says which promise broke, what was expected and what actually happened; most of the value of a test lies in that message. Undo the change before you continue.

> ⚠ **THINK.** Python also has a built-in `assert` statement: `assert rover.age == 6, "Age is now 6"` raises an `AssertionError` if the condition is false. It is fine inside a test, and pytest (Section 6.7) is built on it. But never use `assert` to check *user data* in the program itself: running Python with the `-O` option removes every `assert` statement, and your check silently disappears. Java's `assert` keyword is even weaker — it is switched *off* unless you start the program with `java -ea`. Program logic uses `if` and exceptions; `assert` is for tests and for statements about your own code that "cannot" be false.

6.6 The Anatomy of a Unit Test

Every test in **Listing 6.7**, and every test in the rest of this book, has the same three parts, shown in **Figure 6.2**.

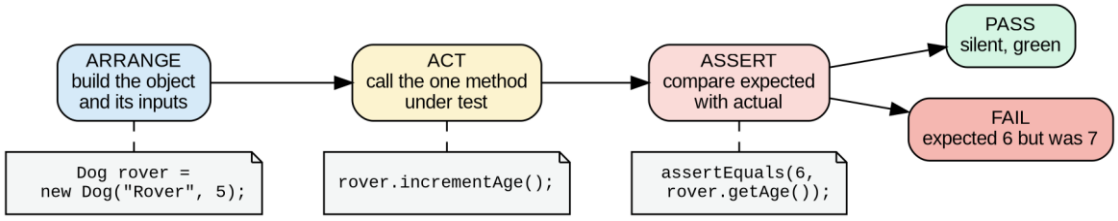


Figure 6.2 Arrange, act, assert: build the object, call the one method under test, compare the expected value with the actual one.

Arrange. Build the object under test and everything it needs: the dog, the order, the account. Keep it minimal; a test that builds twenty objects to check one number is hard to read and breaks for reasons that have nothing to do with the thing it tests.

Act. Call the *one* method you are testing. One test, one behavior. When a test that checks five behaviors fails, you do not yet know which of the five is broken.

Assert. Compare the expected value with the actual value. The expected value is written as a *literal* — `6`, not `5 + 1` and certainly not a copy of the formula the method uses, because a test that recomputes the answer with the same formula will happily agree with the same bug.

The three parts are sometimes called *given, when, then*: *given* a dog of age 5, *when* it has a birthday, *then* its age is 6. Say your test out loud in that form before you write it; if the sentence is awkward, the test probably is too.

6.6.1 Comparing numbers with a fractional part

Integers compare exactly. Floating-point numbers — `double` in Java, `float` in Python — usually do not, because most decimal fractions cannot be stored exactly in binary. The number 0.1 is stored as a binary fraction very slightly different from one tenth, and the small errors add up. **Listing 6.9** and **Listing 6.10** show the classic surprise.

Listing 6.9 `DoubleEquality.java`: why `==` is the wrong test for two doubles.

```java
/**
 * Why two doubles should almost never be compared with ==.
 */
public class DoubleEquality {
    /**
     * Returns true when two doubles differ by at most epsilon.
     *
     * @param expected the value we hoped for
     * @param actual   the value we computed
     * @param epsilon  the largest difference we will tolerate
     * @return whether the two values are close enough
     */
    static boolean close(double expected, double actual, double epsilon) {
        return Math.abs(expected - actual) <= epsilon;
    }

    /**
     * Prints the classic 0.1 + 0.2 surprise and two ways around it.
     *
     * @param args not used
     */
    public static void main(String[] args) {
        double a = 0.1 + 0.2;
        System.out.println("0.1 + 0.2          = " + a);
        System.out.println("a == 0.3           : " + (a == 0.3));
        System.out.println("close(0.3, a, 1e-9): " + close(0.3, a, 1e-9));

        long priceA = 10_000;   // money kept as whole rupiah in a long
        long priceB = 20_000;
        System.out.println("10000 + 20000 == 30000: " + (priceA + priceB == 30_000));
    }
}
```

```
0.1 + 0.2          = 0.30000000000000004
a == 0.3           : false
close(0.3, a, 1e-9): true
10000 + 20000 == 30000: true
```

Listing 6.10 `double_equality.py`: the same experiment, plus `math.isclose`.

```python
"""Why two floats should almost never be compared with ==."""
import math


def close(expected, actual, epsilon):
    """Return True when two floats differ by at most epsilon."""
    return abs(expected - actual) <= epsilon


a = 0.1 + 0.2
print("0.1 + 0.2          =", a)
print("a == 0.3           :", a == 0.3)
print("close(0.3, a, 1e-9):", close(0.3, a, 1e-9))
print("math.isclose(0.3, a):", math.isclose(0.3, a))

price_a = 10_000        # money kept as whole rupiah in an int
price_b = 20_000
print("10000 + 20000 == 30000:", price_a + price_b == 30_000)
```

```
0.1 + 0.2          = 0.30000000000000004
a == 0.3           : False
close(0.3, a, 1e-9): True
math.isclose(0.3, a): True
10000 + 20000 == 30000: True
```

The cure is to accept any answer *close enough* to the expected one. We choose a small tolerance ε (the Greek letter epsilon) and say that the expected value e and the actual value a agree when

$$|e - a| \leq \varepsilon \tag{6.2}$$

where $|e - a|$ is the absolute value of their difference — how far apart they are, ignoring which one is larger. Equation (6.2) is exactly the method `close` in both listings. Here $|0.3 - 0.30000000000000004| = 4 \times 10^{-17}$, far below $\varepsilon = 10^{-9}$, so the test agrees. Every test library has this comparison built in: JUnit's `assertEquals(expected, actual, delta)`, `unittest`'s `assertAlmostEqual(expected, actual, delta=...)`, and pytest's `actual == pytest.approx(expected)`.

Choose ε big enough to swallow rounding noise and small enough to catch a real mistake; for rupiah, 10^{-6} is plenty. Better still for money: **do not use floating point at all**. Keep amounts as whole rupiah in a `long` or a Python `int`, and every comparison is exact, as the pricing engine in Section 6.16 does.

6.7 JUnit 6 and unittest

The hand-made `check` method works, but every project would write its own, slightly different. **Test frameworks** do that job once, well, for everybody. A framework supplies three things: a way to mark a method as a test, a library of assertion methods with good failure messages, and a *runner* — a program that finds all the tests, runs each one on a fresh object, and reports. In Java the standard framework is **JUnit 6** [4] (its engine is called *Jupiter*). Python ships `unittest` [5] in its standard library, and most Python projects also use **pytest** [6].

6.7.1 A JUnit 6 test class

Listing 6.11 tests `Dog` with JUnit 6, the current version [4]. Its programming model, called *Jupiter*, is the one JUnit 5 introduced, so the comment in the listing and the many JUnit 5 tutorials on the web still apply word for word.

Listing 6.11 `DogTest.java`: the `Dog` tests written for JUnit 6.

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

/**
 * JUnit 5 tests for {@link Dog}.
 */
class DogTest {
    private Dog rover;

    @BeforeEach
    void setUp() {
        rover = new Dog("Rover", 5);   // a fresh dog before every single test
    }

    @Test
    @DisplayName("a new dog reports the age it was given")
    void getAgeReturnsInitialAge() {
        assertEquals(5, rover.getAge(), "Age is 5");
    }

    @Test
    @DisplayName("incrementAge adds exactly one year")
    void incrementAgeAddsOneYear() {
        rover.incrementAge();
        assertEquals(6, rover.getAge(), "Age is now 6");
    }

    @Test
    @DisplayName("a negative age is rejected")
    void negativeAgeIsRejected() {
        assertThrows(IllegalArgumentException.class, () -> new Dog("Rex", -1));
    }
}
```

Go through it line by line.

- The `import static ... Assertions.assertEquals` lines let us write `assertEquals(...)` instead of `Assertions.assertEquals(...)`. The other imports bring in the three **annotations** we use. An annotation is a word beginning with `@`, written in front of a method, that tells a tool something about the method without changing what the method does. (You have seen one before: `@Override`.)
- There is **no `main` method**. We do not run a test class ourselves; the JUnit runner does, and it looks for methods marked `@Test`. JUnit 6 does not even need the class or its methods to be `public`.
- `@BeforeEach` marks `setUp`, which JUnit runs *before every test method*. Each test therefore gets its own fresh five-year-old Rover. This matters: `incrementAgeAddsOneYear` changes the dog, and if the tests shared one dog, the result of `getAgeReturnsInitialAge` would depend on which test happened to run first. **Tests must be independent of each other and of their order.**
- `@DisplayName` gives the test a readable sentence for the report. Without it, the report shows the method name — which is why the method names themselves are written as little sentences.
- `assertEquals(expected, actual, message)` is the arrange–act–*assert* step. Put the **expected value first**: when the test fails, JUnit prints "expected: <6> but was: <7>", and if you swap the arguments the message tells a lie.
- `assertThrows(IllegalArgumentException.class, () -> new Dog("Rex", -1))` passes only if running the code throws that kind of exception. The strange-looking `() -> new Dog("Rex", -`

`1)` is a **lambda expression**: a small piece of code packaged up, unrun, and handed to `assertThrows`, which runs it inside its own `try`. For now read the arrow as "here is code for you to run later"; Chapter 10 explains lambdas properly.

> ✓ **KEY INSIGHT.** JUnit is two things at once: a *library* you write tests with (`@Test`, `assertEquals`) and an *application* that runs them. That is why a test class has no `main`: the `main` belongs to the runner.

Many existing projects still use JUnit 4, with the same ideas under different names: `org.junit.Test`, `@Before` instead of `@BeforeEach`, and the **message first** in `assertEquals("Age is 5", 5, rover.getAge())`. Read both; write new tests in JUnit 6.

6.7.2 Running JUnit 6 in VS Code and in a terminal

JUnit is not part of the JDK; it is a *library*, a `.jar` file (a zip of compiled classes). The simplest choice for a beginner is the single file `junit-platform-console-standalone-6.1.3.jar` (any 6.x version works the same), which contains JUnit 6 and its runner. Download it from Maven Central and save it in a folder `lib` next to your `.java` files.

In VS Code (the Extension Pack for Java from Chapter 1 includes the *Test Runner for Java*), open the folder that holds `Dog.java`, `DogTest.java` and `lib`; the extension adds every `.jar` in `lib` to the project (*Java Projects → Referenced Libraries*). A green triangle appears beside the class and beside every `@Test` method; click it, or open the *Testing* view (the flask icon) and click *Run Tests*. Each test gets a green tick or a red cross, and clicking a cross shows the failure message and line. If VS Code offers to "Enable Java Tests", accept and choose *JUnit Jupiter*; it downloads the same jar into `lib`.

In a terminal, in the same folder (on Windows, PowerShell accepts the same commands):

```
javac -cp "lib/*" -d out Dog.java DogTest.java
java -jar lib/junit-platform-console-standalone-6.1.3.jar execute -cp out --select-class
DogTest
```

The first line compiles both classes into `out` with the jar on the *class path* (`-cp`); the second starts JUnit's console runner on the compiled classes. It prints a tree of the tests with a tick or a cross beside each display name and a summary of tests *successful* and *failed*. A failure shows the display name, the message `Age is now 6 ==> expected: <6> but was: <7>`, and the file and line of the assertion.

> ⚠ **THINK.** The listings in this chapter were compiled with `javac` 21 and run on the machine used to prepare the book, which had no internet access and therefore no JUnit jar. The two JUnit classes (`DogTest.java` here and `PriceRuleTest.java` in Section 6.16) were compiled and run against a small stand-in that imitates the few JUnit 6 annotations and assertions they use, and all their tests passed; the verdicts printed in this book come from the plain checkers `DogCheck` and `PriceRuleCheck`, which run the same checks without any library. That is also a lesson: a framework is a convenience, not a requirement. When JUnit is unavailable, a `check` method and a non-zero exit status will still protect you.

6.7.3 unittest

Python's standard library has its own framework, `unittest`, modeled on the JUnit of the early 2000s. **Listing 6.12** is `DogTest` again.

Listing 6.12 `test_dog.py`: the `Dog` tests written for `unittest`.

```
"""unittest tests for Dog."""
import unittest

from dog import Dog


class DogTest(unittest.TestCase):
    def setUp(self):
        self.rover = Dog("Rover", 5)     # a fresh dog before every single test

    def test_age_returns_initial_age(self):
        """a new dog reports the age it was given"""
        self.assertEqual(5, self.rover.age, "Age is 5")

    def test_increment_age_adds_one_year(self):
        """increment_age adds exactly one year"""
        self.rover.increment_age()
        self.assertEqual(6, self.rover.age, "Age is now 6")

    def test_negative_age_is_rejected(self):
        """a negative age is rejected"""
        with self.assertRaises(ValueError):
            Dog("Rex", -1)


if __name__ == "__main__":
    unittest.main()
```

The shape is the Java shape. A test class extends `unittest.TestCase`. Every method whose name starts with `test` is a test — the name *is* the marker, playing the part of `@Test`. `setUp` plays the part of `@BeforeEach`. The first line of a test's docstring plays the part of `@DisplayName`. `self.assertEqual(expected, actual, message)` and `with self.assertRaises(ValueError):` do what JUnit's `assertEquals` and `assertThrows` do; the `with` block replaces the lambda.

Run it from the folder that holds both files with `python3 -m unittest -v test_dog` (the `-v` asks for one line per test):

```
test_age_returns_initial_age (test_dog.DogTest.test_age_returns_initial_age)
a new dog reports the age it was given ... ok
test_increment_age_adds_one_year (test_dog.DogTest.test_increment_age_adds_one_year)
increment_age adds exactly one year ... ok
test_negative_age_is_rejected (test_dog.DogTest.test_negative_age_is_rejected)
a negative age is rejected ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

`python3 -m unittest` with no file name *discovers* every file named `test*.py` in the folder and runs them all. In VS Code, open the *Testing* view, click *Configure Python Tests*, choose `unittest`, the folder `.`, and the pattern `test*.py`; the flask icon then shows the same green ticks as for Java.

And here is the same failure as before — `self._age += 2` planted in `dog.py` — as `unittest` reports it (the long folder name in the `File` line is shortened here):

```
.F.
======================================================================
FAIL: test_increment_age_adds_one_year (test_dog.DogTest.test_increment_age_adds_one_year)
increment_age adds exactly one year
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../test_dog.py", line 18, in test_increment_age_adds_one_year
    self.assertEqual(6, self.rover.age, "Age is now 6")
AssertionError: 6 != 7 : Age is now 6

----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (failures=1)
```

The first line, `.F.`, is the compact progress bar: one character per test, a dot for a pass and `F` for a failure.

6.7.4 pytest

pytest (install it once with `pip install pytest`) drops the ceremony. A test is any function whose name starts with `test_`, in a file whose name starts with `test_`; the assertion is Python's own `assert` statement; no class is required. The pricing tests in Section 6.16 are written that way, and they show pytest's best trick: when an `assert` fails, pytest takes the expression apart and prints the value of every piece. pytest also runs `unittest` classes unchanged, so `python3 -m pytest` in a folder runs both kinds. **Table 6.2** puts the three frameworks side by side.

Table 6.2 The same testing ideas in JUnit 6, `unittest` and pytest.

Purpose	JUnit 6 (Java)	Unittest (Python)	Pytest (Python)
mark a test	`@Test`	method name `test...` in a `TestCase`	function name `test_...`
before each test	`@BeforeEach`	`setUp(self)`	a *fixture* function
equal	`assertEquals(e, a)`	`assertEqual(e, a)`	`assert a == e`
close enough	`assertEquals(e, a, delta)`	`assertAlmostEqual(e, a, delta=d)`	`assert a == pytest.approx(e)`
true / false	`assertTrue(c)`, `assertFalse(c)`	`assertTrue(c)`, `assertFalse(c)`	`assert c`, `assert not c`
throws	`assertThrows(T.class, () -> ...)`	`with self.assertRaises(T):`	`with pytest.raises(T):`
run	VS Code, or the console launcher	`python -m unittest`	`python -m pytest`

6.8 Choosing Test Cases, and Writing Them First

A test suite is only as good as the inputs it tries. Trying `3.2` kilograms ten times proves no more than trying it once. Good testers choose inputs systematically, with two classic techniques and one habit [7].

6.8.1 Equivalence classes and boundaries

Divide the possible inputs into **equivalence classes**: groups for which the code should behave in the same way, so that one representative of each group is as good as any other. Then test the

boundaries between the groups, because that is where programmers write `<` when they meant `<=`.

Take the laundry's per-kilogram price, which Section 6.16 will build: wash and fold costs 7 000 rupiah per kilogram, every *started* kilogram counts (3.2 kg is charged as 4 kg), and nobody pays for less than 3 kg. The weights fall into four classes, and **Table 6.3** picks the tests.

Table 6.3 Equivalence classes and boundary values for the per-kilogram price with a 3 kg minimum.

Class of input	Representative	Boundary values	Expected
no weight (invalid)	0	0	exception
up to the minimum	2.0	3.0	21 000 (3 kg)
just past the minimum	3.2	3.01	28 000 (4 kg)
well above	5.5	—	42 000 (6 kg)

The same thinking applies to everything you test. For an array, test the first element, the last element, and an empty array — index errors live at the ends. For every `if`, make sure some test takes the *true* branch and some test takes the *false* branch. For every exception a method promises in its documentation, write a test that provokes it. For a `double`, compare with a tolerance (Equation (6.2)).

6.8.2 Test first

The habit is the surprising one: **write the test before the code**. The practice is called *test-driven development* (TDD) [8], and it runs in a short loop, shown in **Figure 6.3**.

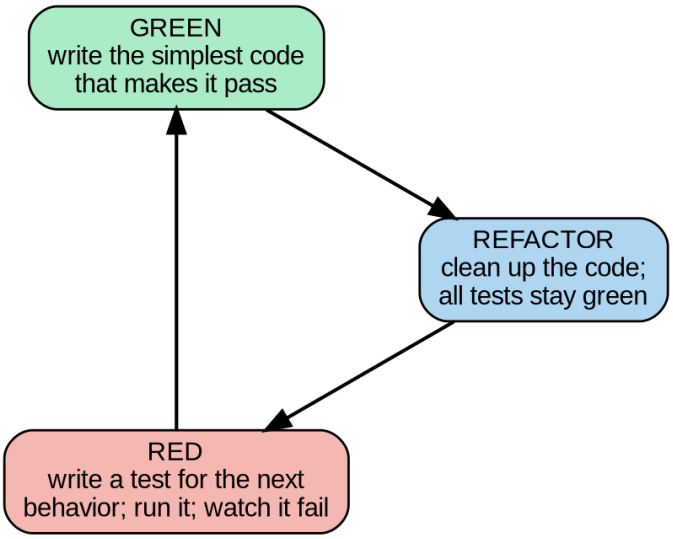


Figure 6.3 The test-first cycle: a failing test (red), the simplest code that passes (green), then tidying with the tests as a safety net (refactor).

Red. Write one test for the next small behavior you want, run it, and *watch it fail*. The failure proves that the test can fail — a test that passes before the code exists is testing nothing.

Green. Write the simplest code that makes the test pass. Not the cleverest; the simplest.

Refactor. Now improve the code — rename, remove duplication, simplify — running the tests after every change. They stay green, so you know the improvements changed nothing that matters.

Writing the test first forces you to decide what the method should *do*, from the caller's side, before you are distracted by how. Exercise 6.6 walks through one complete cycle for a loyalty card.

6.9 When Are Two Objects Equal?

Sari walks into the laundry on Monday and the cashier types her in as "Sari", phone `0812-555-0101`. On Thursday a different cashier types " Sari ", `0812 555 0101`. Is that the same customer? To a person, obviously yes. To a program, it depends entirely on what the class says — and if the class says nothing, the answer is *no*, and Sari's loyalty points end up split between two customers.

6.9.1 Identity and equality

Every language with objects has two different questions that sound alike.

> **DEFINITION.** Two references are *identical* when they point to the very same object in memory. Two objects are *equal* when they represent the same value, according to a rule the class defines. In Java, `a == b` tests identity for objects and `a.equals(b)` tests equality. In Python, `a is b` tests identity and `a == b` tests equality (by calling `a.__eq__(b)`).

Figure 6.4 draws the situation: two variables, two separate objects, identical contents.

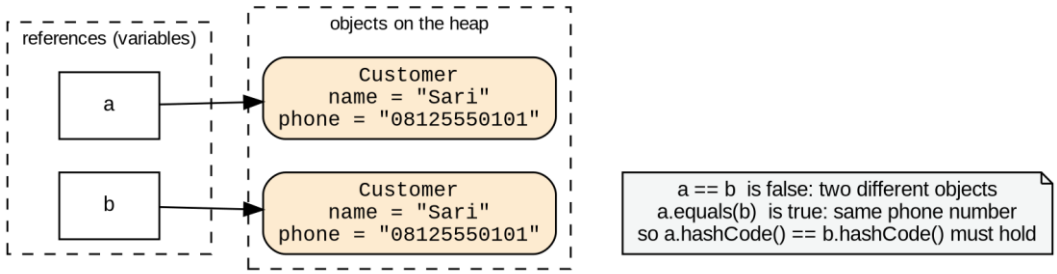


Figure 6.4 Two references to two different objects with the same contents: not identical, but equal if the class says so — and then their hash codes must be equal too.

What does a class say if we write nothing? Every Java class inherits `equals` from the class `Object`, and `Object.equals` is simply `==`: an object is equal only to itself [9]. Python's default `__eq__` behaves the same way. **Listing 6.13** shows the consequences, and then shows the second trap: a class that defines `equals` but forgets `hashCode`.

Listing 6.13 `IdentityDemo.java`: a class with no `equals`, and a class with `equals` but no `hashCode`.

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

/** A customer class that overrides nothing: it inherits equals, hashCode and toString from
Object. */
class PlainCustomer {
    final String name;
    final String phone;

    PlainCustomer(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }
}

/** A customer class that overrides equals but forgets hashCode. */
class HalfCustomer {
    final String name;
    final String phone;

    HalfCustomer(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }

    @Override
    public boolean equals(Object other) {
        if (!(other instanceof HalfCustomer)) {
            return false;
        }
        HalfCustomer that = (HalfCustomer) other;
        return phone.equals(that.phone);
    }
}
```

Listing 6.13 (Continued) *IdentityDemo.java: a class with no equals, and a class with equals but no hashCode.*

```
/**
 * What happens to equality when a class does not define it properly.
 */
public class IdentityDemo {
    /**
     * Hides the memory-dependent hexadecimal part of a default toString.
     *
     * @param o any object
     * @return its toString with the hexadecimal hash replaced by a placeholder
     */
    static String masked(Object o) {
        return o.toString().replaceAll("@[0-9a-f]+", "@<hash>");
    }

    /**
     * Compares two customers typed in twice from the same data.
     *
     * @param args not used
     */
    public static void main(String[] args) {
        PlainCustomer a = new PlainCustomer("Sari", "0812-555-0101");
        PlainCustomer b = new PlainCustomer("Sari", "0812-555-0101");
        System.out.println("PlainCustomer");
        System.out.println("  a == b      : " + (a == b));
        System.out.println("  a.equals(b) : " + a.equals(b));
        System.out.println("  a prints as : " + masked(a));
        Set<PlainCustomer> plainSet = new HashSet<>();
        plainSet.add(a);
        plainSet.add(b);
        System.out.println("  set size    : " + plainSet.size());

        HalfCustomer c = new HalfCustomer("Sari", "0812-555-0101");
        HalfCustomer d = new HalfCustomer("Sari", "0812-555-0101");
        System.out.println("HalfCustomer (equals without hashCode)");
        System.out.println("  c.equals(d) : " + c.equals(d));
        List<HalfCustomer> list = new ArrayList<>();
        list.add(c);
        System.out.println("  list contains d: " + list.contains(d));   // List uses equals
only
        Set<HalfCustomer> halfSet = new HashSet<>();
        halfSet.add(c);
        System.out.println("  set contains d : " + halfSet.contains(d)); // HashSet uses
hashCode first
        halfSet.add(d);
        System.out.println("  set size       : " + halfSet.size());
    }
}
```

```
PlainCustomer
  a == b      : false
  a.equals(b) : false
  a prints as : PlainCustomer@<hash>
  set size    : 2
HalfCustomer (equals without hashCode)
  c.equals(d) : true
  list contains d: true
  set contains d : false
  set size       : 2
```

The first block is what we feared: two `PlainCustomer` objects typed from the same data are neither identical nor equal, and a `HashSet` holds both. `PlainCustomer@<hash>` is the default `toString` from `Object`: the class name and a hexadecimal number that changes from run to run (the program prints `<hash>` so that the output here matches yours).

The second block is worse, because it looks fixed. `HalfCustomer` overrides `equals`, `c.equals(d)` is `true`, and an `ArrayList` finds `d`. But the `HashSet` says `d` is not there, and after adding `d` it holds two elements.

The reason is how a hash set works. It does not compare a new element with every element it holds; it first asks for a **hash code**, an `int` computed from the object, and uses it to pick one small *bucket*. Only the elements in that bucket are compared with `equals`. `HalfCustomer` inherited `hashCode` from `Object`, which gives different objects different numbers, so `c` and `d` land in different buckets and `equals` is never called. `javac -Xlint:all IdentityDemo.java` warns about exactly this:

```
IdentityDemo.java:18: warning: [overrides] Class HalfCustomer overrides equals, but neither
it nor any superclass overrides hashCode method
class HalfCustomer {
^
1 warning
```

The rule that makes hash sets and hash maps work is short. For any two objects a and b and the hash function h,

$$a \text{ equals } b \;\Rightarrow\; h(a) = h(b) \tag{6.3}$$

The arrow ⇒ reads “implies”: *if* two objects are equal, *then* their hash codes must be equal. Equation (6.3) does not work backwards — two different objects may share a hash code (a *collision*), and that only costs a little speed, because `equals` sorts them out inside the bucket. The rule is broken only when equal objects get different codes, and then sets and maps stop working. The practical consequence: **whenever you override `equals`, override `hashCode` too, using the same fields.**

Python is stricter here, which is kinder. **Listing 6.14** repeats the experiment.

Listing 6.14 `identity_demo.py`: the same two mistakes in Python; the second one is caught.

```
"""What happens to equality when a class does not define it properly."""
import re


class PlainCustomer:
    """A customer class that defines nothing: it inherits object's behavior."""

    def __init__(self, name, phone):
        self.name = name
        self.phone = phone


class HalfCustomer:
    """A customer class that defines __eq__ but not __hash__."""

    def __init__(self, name, phone):
        self.name = name
        self.phone = phone

    def __eq__(self, other):
        if not isinstance(other, HalfCustomer):
            return NotImplemented
        return self.phone == other.phone


def masked(o):
    """Hide the memory address in a default repr."""
    return re.sub(r"0x[0-9a-fA-F]+", "0x<address>", repr(o))


a = PlainCustomer("Sari", "0812-555-0101")
b = PlainCustomer("Sari", "0812-555-0101")
print("PlainCustomer")
print("  a is b      :", a is b)
print("  a == b      :", a == b)
print("  a prints as :", masked(a))
print("  set size    :", len({a, b}))
```

Listing 6.14 (Continued) *identity_demo.py: the same two mistakes in Python; the second one is caught.*

```
c = HalfCustomer("Sari", "0812-555-0101")
d = HalfCustomer("Sari", "0812-555-0101")
print("HalfCustomer (__eq__ without __hash__)")
print("  c == d      :", c == d)
print("  d in [c]    :", d in [c])          # a list uses == only
print("  __hash__ is :", HalfCustomer.__hash__)
try:
    {c, d}                                  # a set needs hash()
except TypeError as e:
    print("  set fails   :", e)
```

```
PlainCustomer
  a is b      : False
  a == b      : False
  a prints as : <__main__.PlainCustomer object at 0x<address>>
  set size    : 2
HalfCustomer (__eq__ without __hash__)
  c == d      : True
  d in [c]    : True
  __hash__ is : None
  set fails   : unhashable type: 'HalfCustomer'
```

When a Python class defines `__eq__` and not `__hash__`, Python sets `__hash__` to `None`, which makes the objects **unhashable**: they cannot go into a set or be used as dictionary keys at all. You

get a clear `TypeError` on the spot instead of a silent wrong answer. (Newer versions of Python word the message a little differently; the meaning is the same.)

6.9.2 Writing equals and hashCode properly

Listing 6.15 is the `Customer` class done right. A customer is identified by phone number, so the constructor first *normalizes* the number — keeps only the digits — and `equals` and `hashCode` then use the normalized phone and nothing else. The name is ignored on purpose: “Sari” and “Sari W.” with the same phone are the same person.

Listing 6.15 `Customer.java`: equality by phone number, a matching `hashCode`, and a readable `toString`.

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

/**
 * A laundry customer, identified by phone number.
 *
 * <p>Two {@code Customer} objects are equal when their phone numbers
 * have the same digits, however the numbers were typed.</p>
 */
public final class Customer {
    private final String name;
    private final String phone;   // digits only, e.g. "08125550101"

    /**
     * Creates a customer.
     *
     * @param name  the customer's name as spoken at the counter
     * @param phone the phone number in any format, e.g. "0812-555-0101"
     */
    public Customer(String name, String phone) {
        this.name = name.strip();
        this.phone = phone.replaceAll("[^0-9]", "");   // keep the digits only
    }

    /**
     * Gets the name.
     *
     * @return the name
     */
    public String getName() {
        return name;
    }

    /**
     * Gets the phone number.
     *
     * @return the phone number, digits only
     */
    public String getPhone() {
        return phone;
    }
```

Listing 6.15 (Continued) *Customer.java: equality by phone number, a matching hashCode, and a readable toString.*

```
    /**
     * Two customers are equal when they have the same phone number.
     *
     * @param other the object to compare with
     * @return true if {@code other} is a Customer with the same phone number
     */
    @Override
    public boolean equals(Object other) {
        if (this == other) {
            return true;                        // the very same object
        }
        if (!(other instanceof Customer)) {
            return false;                       // null, or some other class
        }
        Customer that = (Customer) other;
        return phone.equals(that.phone);
    }

    /**
     * Equal customers must have equal hash codes, so hash the phone number only.
     *
     * @return a hash code built from the phone number
     */
    @Override
    public int hashCode() {
        return phone.hashCode();
    }

    /**
     * Describes the customer for logs and debugging.
     *
     * @return for example {@code Customer[name=Sari, phone=08125550101]}
     */
    @Override
    public String toString() {
        return "Customer[name=" + name + ", phone=" + phone + "]";
    }
```

Listing 6.15 (Continued) *Customer.java: equality by phone number, a matching hashCode, and a readable toString.*

```
    /**
     * Shows equals, hashCode and toString at work.
     *
     * @param args not used
     */
    public static void main(String[] args) {
        Customer a = new Customer("Sari", "0812-555-0101");
        Customer b = new Customer(" Sari ", "0812 555 0101");
        Customer budi = new Customer("Budi", "0813-222-0202");
        System.out.println(a);
        System.out.println(b);
        System.out.println("a == b      : " + (a == b));
        System.out.println("a.equals(b) : " + a.equals(b));
        System.out.println("same hash   : " + (a.hashCode() == b.hashCode()));
        System.out.println("a.equals(budi): " + a.equals(budi));
        System.out.println("a.equals(null): " + a.equals(null));

        Set<Customer> customers = new HashSet<>();
        customers.add(a);
        customers.add(b);
        customers.add(budi);
        System.out.println("distinct customers: " + customers.size());

        Map<Customer, Integer> visits = new HashMap<>();
        Customer[] counter = {a, budi, b, a};   // four visits typed at the counter
        for (Customer c : counter) {
            visits.put(c, visits.getOrDefault(c, 0) + 1);
        }
        System.out.println("visits by Sari: " + visits.get(new Customer("Sari",
"08125550101")));
        System.out.println("visits by Budi: " + visits.get(budi));
    }
}
```

```
Customer[name=Sari, phone=08125550101]
Customer[name=Sari, phone=08125550101]
a == b      : false
a.equals(b) : true
same hash   : true
a.equals(budi): false
a.equals(null): false
distinct customers: 2
visits by Sari: 3
visits by Budi: 1
```

Read `equals` line by line, because the same four steps appear in almost every `equals` you will write.

1. `if (this == other) return true;` — an object is always equal to itself. This is also a quick exit for the common case.
2. `if (!(other instanceof Customer)) return false;` — the parameter has type `Object`, so anything at all can arrive: a `String`, a `Dog`, or `null`. `instanceof` answers "is this object a `Customer` (or a subclass of one)?", and it answers `false` for `null`, so this one line handles both cases. Never write `equals(Customer other)`: that does not *override* `Object.equals(Object)`, it *overloads* it (Section 6.11), and collections will keep calling the old one. The `@Override` annotation catches exactly this mistake at compile time.
3. `Customer that = (Customer) other;` — now that we know it is a `Customer`, a *cast* tells the compiler to treat it as one, so that we may read `that.phone`. Section 6.14.1 says more about casts.

4. `return phone.equals(that.phone);` — compare the fields that define equality. Fields that are objects are compared with `equals`, never `==`: two `String` objects with the same characters are equal but not necessarily identical.

`hashCode` returns `phone.hashCode()` — the hash of exactly the field `equals` compares. If equality used two fields, `Objects.hash(field1, field2)` combines them (Exercise 6.7 does this).

The output confirms every promise: `a == b` is `false` but `a.equals(b)` is `true`; the hash codes agree; the set holds two distinct customers, not three; and the visit counter, a `HashMap` from customer to visits, counts Sari three times although she was typed two ways.

The rules Java requires of every `equals` [9], [10] are worth stating once, for any non-null references `x`, `y` and `z`:

> **DEFINITION (the equals contract).** *Reflexive*: `x.equals(x)` is true. *Symmetric*: `x.equals(y)` is true exactly when `y.equals(x)` is true. *Transitive*: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`. *Consistent*: repeated calls give the same answer while the objects do not change. *Null*: `x.equals(null)` is false. And, from Equation (6.3): equal objects have equal hash codes.

The Python version, **Listing 6.16**, follows the same plan.

Listing 6.16 `customer.py`: `__eq__`, a matching `__hash__`, and both `__repr__` and `__str__`.

```
"""A laundry customer, identified by phone number."""
import re


class Customer:
    """A laundry customer, identified by phone number.

    Two Customer objects are equal when their phone numbers have the
    same digits, however the numbers were typed.
    """

    def __init__(self, name, phone):
        """Create a customer.

        Args:
            name: the customer's name as spoken at the counter.
            phone: the phone number in any format, e.g. "0812-555-0101".
        """
        self._name = name.strip()
        self._phone = re.sub(r"[^0-9]", "", phone)   # keep the digits only

    @property
    def name(self):
        """The name."""
        return self._name

    @property
    def phone(self):
        """The phone number, digits only."""
        return self._phone

    def __eq__(self, other):
        """Two customers are equal when they have the same phone number."""
        if self is other:
            return True                     # the very same object
        if not isinstance(other, Customer):
            return NotImplemented           # let Python try the other side, then say False
        return self._phone == other._phone

    def __hash__(self):
        """Equal customers must have equal hashes, so hash the phone number only."""
        return hash(self._phone)
```

Listing 6.16 (Continued) *customer.py: __eq__, a matching __hash__, and both __repr__ and __str__.*

```
    def __repr__(self):
        """The unambiguous form, for developers: looks like the constructor call."""
        return f"Customer(name={self._name!r}, phone={self._phone!r})"

    def __str__(self):
        """The friendly form, for receipts and messages."""
        return f"{self._name} ({self._phone})"


if __name__ == "__main__":
    a = Customer("Sari", "0812-555-0101")
    b = Customer(" Sari ", "0812 555 0101")
    budi = Customer("Budi", "0813-222-0202")
    print(a)                    # print() uses __str__
    print(repr(b))              # repr() uses __repr__
    print([a, budi])            # a list shows its items with __repr__
    print("a is b      :", a is b)
    print("a == b      :", a == b)
    print("same hash   :", hash(a) == hash(b))
    print("a == budi   :", a == budi)
    print("a == None   :", a == None)

    customers = {a, b, budi}
    print("distinct customers:", len(customers))

    visits = {}
    for c in [a, budi, b, a]:   # four visits typed at the counter
        visits[c] = visits.get(c, 0) + 1
    print("visits by Sari:", visits[Customer("Sari", "08125550101")])
    print("visits by Budi:", visits[budi])
```

```
Sari (08125550101)
Customer(name='Sari', phone='08125550101')
[Customer(name='Sari', phone='08125550101'), Customer(name='Budi', phone='08132220202')]
a is b      : False
a == b      : True
same hash   : True
a == budi   : False
a == None   : False
distinct customers: 2
visits by Sari: 3
visits by Budi: 1
```

Two Python details differ from Java. First, when the other object is not a `Customer`, `__eq__` returns the special value `NotImplemented` rather than `False`. That tells Python "I don't know how to compare with that"; Python then gives the other object a chance to answer, and if it cannot either, the result of `==` is `False` — which is why `a == None` printed `False`. Second, `__hash__` hashes the same field `__eq__` compares, which keeps Equation (6.3) true.

> ✓ **KEY INSIGHT.** You rarely need to write these methods by hand for a class that is *just data*. A Java **record** — `record CustomerRecord(String name, String phone) {}` — gets `equals`, `hashCode` and `toString` written by the compiler, and a Python `@dataclass(frozen=True)` gets `__eq__`, `__hash__` and `__repr__`. Both compare *every* field, though, so they would treat "Sari" and "Sari W." as different people. When equality means something narrower than "all fields equal" — as for `Customer` — you write the methods yourself. Exercise 6.8 compares the two approaches.

6.10 Describing an Object: toString, str and repr

When Java needs an object as text — in `System.out.println(x)`, in `"Customer: " + x`, or in a debugger — it calls `x.toString()`. The version inherited from `Object` tells a human almost nothing, as `IdentityDemo` showed, so override it in every class you might print. The conventional shape, used by records and imitated in **Listing 6.15**, is `ClassName[field=value, field=value]`.

Python splits the job in two, and **Listing 6.16** writes both:

- `__repr__` is the **unambiguous, developer-facing** form. Ideally it looks like the expression that would rebuild the object, as in `Customer(name='Sari', phone='08125550101')`. The interactive prompt, the debugger, `repr(x)`, and a list printing its items all use it. (The `!r` inside the f-string asks for the `repr` of the field, which is where the quotes around `'Sari'` come from.)
- `__str__` is the **friendly, user-facing** form, used by `print(x)` and `str(x)`: `Sari (08125550101)`, suitable for a receipt.

If a class defines only `__repr__`, `str()` falls back to it. In the output of **Listing 6.16**, `print(a)` used `__str__`, `print(repr(b))` used `__repr__`, and printing a *list* used `__repr__` for each element.

Java has only `toString`, so write the developer's form there and a separately named method, such as `receiptLine()`, for a customer-facing format, as the laundry program in Section 6.16 does.

> ⚠ **TRAP.** Never put secrets into `toString` or `__repr__`. These methods end up in log files, error reports and debugger windows. A customer's full phone number is borderline; a password, an access token or a full identity-card number must never appear there. Under Indonesia's personal-data protection law (UU PDP 27/2022), careless logging of personal data is a real legal problem, not a style issue.

6.11 Many Versions of a Method: Overloading

Here is a small, real annoyance. We need the average balance of an array of bank accounts, and elsewhere the average area of an array of countries. **Listing 6.17** writes both.

Listing 6.17 `AverageTwice.java`: two methods with the same name and almost the same body.

```
/**
 * Two nearly identical average methods: the duplication that interfaces remove.
 */
public class AverageTwice {
    /** A bank account; nested here so its name cannot clash with other files. */
    static class Account {
        private final String id;
        private final String owner;
        private final double balance;

        Account(String id, String owner, double balance) {
            this.id = id;
            this.owner = owner;
            this.balance = balance;
        }

        double getBalance() {
            return balance;
        }
    }

    /** A country with an area in square kilometers. */
    static class Country {
        private final String name;
        private final double area;

        Country(String name, double area) {
            this.name = name;
            this.area = area;
        }

        double getArea() {
            return area;
        }
    }
```

Listing 6.17 (Continued) *AverageTwice.java: two methods with the same name and almost the same body.*

```
    /**
     * Computes the average balance of some accounts.
     *
     * @param objects the accounts
     * @return the average balance, or 0 for an empty array
     */
    public static double average(Account[] objects) {
        if (objects.length == 0) {
            return 0;
        }
        double sum = 0.0;
        for (Account obj : objects) {       // loops through each object in the array
            sum = sum + obj.getBalance();
        }
        return sum / objects.length;
    }

    /**
     * Computes the average area of some countries.
     *
     * @param objects the countries
     * @return the average area, or 0 for an empty array
     */
    public static double average(Country[] objects) {
        if (objects.length == 0) {
            return 0;
        }
        double sum = 0.0;
        for (Country obj : objects) {       // the same loop, a different getter
            sum = sum + obj.getArea();
        }
        return sum / objects.length;
    }

    /**
     * Calls both versions of average.
     *
     * @param args not used
     */
    public static void main(String[] args) {
        Account[] accounts = {
            new Account("01", "Aleksandra", 100),
            new Account("02", "Natasha", 150),
            new Account("03", "Sergei", 125)
        };
        Country[] countries = {
```

Listing 6.17 (Continued) *AverageTwice.java: two methods with the same name and almost the same body.*

```
            new Country("Indonesia", 1_904_569),
            new Country("Malaysia", 330_803),
            new Country("Singapore", 734)
        };
        System.out.println("Average balance is: " + average(accounts));
        System.out.println("Average area is: " + average(countries));
    }
}
```

```
Average balance is: 125.0
Average area is: 745368.6666666666
```

(`Account` and `Country` are written *inside* `AverageTwice` as `static class` declarations so that their names cannot clash with the better versions in Section 6.12. For now, read a nested static class simply as a helper class that lives inside another class's file.)

One thing here is good: both methods are called `average`, and Java has no trouble telling them apart. This is **method overloading**.

> **DEFINITION.** A method name is *overloaded* when a class declares several methods with that name and different *parameter lists* — a different number of parameters, or different types in some position. The name together with the parameter types is the method's *signature*, and signatures must be unique within a class. The compiler chooses which version to call by looking at the types of the arguments at the call site.

You have already overloaded: the two `BankAccount` constructors are overloads, and `System.out.println` has ten versions, for `int`, `double`, `String`, `Object` and so on. The return type is **not** part of the signature: two methods that differ only in what they return cannot live in one class, because at a call such as `price(3);` the compiler could not choose. `javac` refuses:

```
Oops.java:3: error: method price(int) is already defined in class Oops
    static long price(int kg) { return kg * 7000L; }
                ^
1 error
```

The bad thing in **Listing 6.17** is the duplication: the two bodies are identical apart from one type and one getter name. Overloading let us *name* them the same; it did nothing to let us *write* them once. Section 6.12 fixes that with an interface. First, let us see precisely how Java chooses between overloads, and then how Python, which cannot overload at all, gets the same job done.

6.11.1 How Java chooses an overload

Listing 6.18 declares three versions of `describe` and calls them with six different arguments.

Listing 6.18 `OverloadDemo.java`: which version of an overloaded method does each call choose?

```java
/**
 * Method overloading: one name, several parameter lists.
 */
public class OverloadDemo {
    static String describe(int pieces) {
        return "int version    -> " + pieces + " pieces";
    }

    static String describe(double kilos) {
        return "double version -> " + kilos + " kg";
    }

    static String describe(String customer, int pieces) {
        return "(String, int)  -> " + customer + " brings " + pieces + " pieces";
    }

    /**
     * Calls describe with arguments of different types and shows which version runs.
     *
     * @param args not used
     */
    public static void main(String[] args) {
        System.out.println("describe(3)         : " + describe(3));
        System.out.println("describe(3.5)       : " + describe(3.5));
        System.out.println("describe(\"Sari\", 4) : " + describe("Sari", 4));
        byte small = 2;
        System.out.println("describe(small)     : " + describe(small));  // byte widens to
int
        System.out.println("describe(3L)        : " + describe(3L));     // long widens to
double!
        System.out.println("describe('A')       : " + describe('A'));    // char widens to
int!
    }
}
```

```
describe(3)         : int version    -> 3 pieces
describe(3.5)       : double version -> 3.5 kg
describe("Sari", 4) : (String, int)  -> Sari brings 4 pieces
describe(small)     : int version    -> 2 pieces
describe(3L)        : double version -> 3.0 kg
describe('A')       : int version    -> 65 pieces
```

The first three lines are no surprise: an `int` goes to `describe(int)`, a `double` to `describe(double)`, and two arguments to the two-parameter version. The last three show the rule the compiler actually follows [11]. If no version matches the argument types *exactly*, the compiler looks for one it can reach by **widening** the argument — converting it to a type that can hold every value of the original. A `byte` widens to `int`, so `describe(small)` picks the `int` version. A `long` cannot become an `int` (it might not fit) but it *can* widen to `double`, so `describe(3L)` quietly picks the kilograms version and prints `3.0 kg`. And a `char` is, to Java, a small unsigned number — the code of the character — so `describe('A')` widens `'A'` to the `int` 65 and reports 65 pieces. Only if widening fails does the compiler try *boxing* (turning `int` into `Integer`), and only after that does it try methods with a variable number of arguments. When two versions are equally good, the call is ambiguous and does not compile.

The lesson is not to memorize these phases but to avoid designs that depend on them: overload only when the versions do *the same thing* for different kinds of input. `describePieces` and `describeKilos` would have made both surprises impossible.

6.11.2 Python has no overloading — and does not need it

Python looks up a method by *name* only. A `def` statement simply assigns a function to a name, so a second `def average` replaces the first, exactly as a second `x = ...` replaces the first value of `x`. **Listing 6.19** tries to copy **Listing 6.17** literally.

Listing 6.19 `average_twice.py`: in Python, the second `def average` replaces the first.

```
"""Python cannot keep two functions with the same name: the second replaces the first."""


class Account:
    def __init__(self, id, owner, balance):
        self.id = id
        self.owner = owner
        self.balance = balance


class Country:
    def __init__(self, name, area):
        self.name = name
        self.area = area


def average(objects):
    """Average balance of some accounts."""
    if len(objects) == 0:
        return 0
    total = 0.0
    for obj in objects:
        total = total + obj.balance
    return total / len(objects)


def average(objects):          # the same name again: this definition REPLACES the first
    """Average area of some countries."""
    if len(objects) == 0:
        return 0
    total = 0.0
    for obj in objects:
        total = total + obj.area
    return total / len(objects)
```

Listing 6.19 (Continued) *average_twice.py: in Python, the second def average replaces the first.*

```
accounts = [Account("01", "Aleksandra", 100),
            Account("02", "Natasha", 150),
            Account("03", "Sergei", 125)]
countries = [Country("Indonesia", 1_904_569),
             Country("Malaysia", 330_803),
             Country("Singapore", 734)]
print("Average area is:", average(countries))
try:
    print("Average balance is:", average(accounts))
except AttributeError as e:
    print("Average balance failed:", e)
```

```
Average area is: 745368.6666666666
Average balance failed: 'Account' object has no attribute 'area'
```

The countries work, the accounts fail: only the second function exists, and nothing is reported until the wrong kind of object meets it. (The type checker mypy does catch it, with `Name "average" already defined on line 17`.)

Python covers the uses of overloading with three other tools, all in **Listing 6.20**.

Listing 6.20 `overload_demo.py`: default arguments, `*args` and `functools.singledispatch`.

```
"""Python's answers to overloading: default arguments, *args and singledispatch."""
from functools import singledispatch


# 1. Default arguments: one function, optional parameters
def describe_order(pieces, customer=None):
    if customer is None:
        return f"{pieces} pieces"
    return f"{customer} brings {pieces} pieces"


# 2. *args: any number of positional arguments, collected into a tuple
def total(*prices):
    return sum(prices)


# 3. singledispatch: choose an implementation by the type of the FIRST argument
@singledispatch
def describe(value):
    return f"general version -> {value!r}"


@describe.register
def _(value: int):
    return f"int version     -> {value} pieces"


@describe.register
def _(value: float):
    return f"float version   -> {value} kg"


print("describe_order(3)        :", describe_order(3))
print("describe_order(4, 'Sari'):", describe_order(4, "Sari"))
print("total()                  :", total())
print("total(7000, 21000, 35000):", total(7000, 21000, 35000))
print("describe(3)              :", describe(3))
print("describe(3.5)            :", describe(3.5))
print("describe('Sari')         :", describe("Sari"))
print("describe(True)           :", describe(True))    # bool is a subclass of int!
```

```
describe_order(3)        : 3 pieces
describe_order(4, 'Sari'): Sari brings 4 pieces
total()                  : 0
total(7000, 21000, 35000): 63000
describe(3)              : int version     -> 3 pieces
describe(3.5)            : float version   -> 3.5 kg
describe('Sari')         : general version -> 'Sari'
describe(True)           : int version     -> True pieces
```

Default arguments cover the commonest case, where the versions differ by leaving something out, as with `BankAccount` in **Listing 6.2**. **`*args`** covers "any number of arguments": the star collects all the positional arguments into a tuple, so `total()` with none and `total(7000, 21000, 35000)` with three are the same function. And when the behavior genuinely depends on the *type* of an argument, **`functools.singledispatch`** [12] provides overloading by type for the first argument: `@singledispatch` marks the general version, and `@describe.register` adds a version for each type named in the parameter's annotation. Look at the last line: `describe(True)` chose the `int` version, because in Python `bool` is a subclass of `int`. It is the same trap as Java's `char` becoming 65, in a different language. **Table 6.4** summarizes.

Table 6.4 Overloading in Java and the Python tools that do the same jobs.

Need	Java	Python
optional parameters	overloads that call each other	default arguments
several constructors	overloaded constructors, `this(...)`	one `__init__` with defaults, or `@classmethod` factories
any number of arguments	varargs, `int... values`	`*args` (and `**kwargs` for named ones)
different code per argument type	overloading, chosen at compile time	`functools.singledispatch`, chosen at run time
same code for many types	an interface (Section 6.12)	a protocol or ABC (Section 6.15)

The last row is the one that matters most. Neither overloading nor `singledispatch` removes the duplication of `average`. For that we need a way to say "any object that can give me a number" — and that is what an interface is.

6.12 Interfaces

Look once more at the two `average` methods. What does the loop actually need from each object? Not that it is an account; not that it is a country. It needs exactly one thing: *a number to add up*. If we could write a type that means "an object that can give me its number", one `average` would serve accounts, countries, and anything we invent next year.

That type is an **interface**.

> **DEFINITION.** An *interface* is a named list of method headers — names, parameter types and return types — with no bodies. It describes *what* an object can do and says nothing about *how*. A class *implements* an interface by declaring so (`implements Measurable` in Java) and by supplying a body for every method the interface lists. An interface can then be used as a *type*: a variable, parameter or array of that type can hold any object of any class that implements it.

In Java an interface is declared with the keyword `interface`. **Listing 6.21** is the whole of `Measurable`.

Listing 6.21 `Measurable.java`: an interface with a single method and no code.

```
/**
 * Anything that can be reduced to one number for averaging.
 */
public interface Measurable {
    /**
     * Returns the number that represents this object when averaging.
     *
     * @return the measure
     */
    double getMeasure();      // no body: every implementing class must supply one
}
```

The method has a header and a semicolon where the body would be. Every method declared this way in an interface is automatically `public` and `abstract` — abstract meaning "declared here, implemented elsewhere" — so the two keywords are usually left out. An interface normally lives in a file of its own, named after it, exactly like a public class.

Listing 6.22 makes two classes implement it and writes `average` once.

Listing 6.22 `MeasurableDemo.java`: two classes implement `Measurable`, and one `average` method serves both.

```
/** A bank account, measured by its balance. */
class Account implements Measurable {
    private final String id;
    private final String owner;
    private final double balance;

    Account(String id, String owner, double balance) {
        this.id = id;
        this.owner = owner;
        this.balance = balance;
    }

    @Override
    public double getMeasure() {
        return balance;
    }
}

/** A country, measured by its area. */
class Country implements Measurable {
    private final String name;
    private final double area;

    Country(String name, double area) {
        this.name = name;
        this.area = area;
    }

    @Override
    public double getMeasure() {
        return area;
    }
}

/**
 * One average method for every Measurable class.
 */
public class MeasurableDemo {
    /**
     * Computes the average measure of some objects.
     *
     * @param objects anything that implements Measurable
     * @return the average measure, or 0 for an empty array
     */
    public static double average(Measurable[] objects) {
        if (objects.length == 0) {
```

Listing 6.22 (Continued) *MeasurableDemo.java: two classes implement Measurable, and one average method serves both.*

```
            return 0;
        }
        double sum = 0;
        for (Measurable obj : objects) {     // obj may be an Account or a Country
            sum = sum + obj.getMeasure();
        }
        return sum / objects.length;
    }

    /**
     * Averages accounts two ways, then countries.
     *
     * @param args not used
     */
    public static void main(String[] args) {
        // One way: an array whose element type is the interface
        Measurable[] accounts = new Measurable[3];
        accounts[0] = new Account("01", "Aleksandra", 100);
        accounts[1] = new Account("02", "Natasha", 150);
        accounts[2] = new Account("03", "Sergei", 125);
        System.out.println("Average balance is: " + average(accounts));

        // Second way: an array of the class itself
        Account b1 = new Account("01", "Aleksandra", 100);
        Account b2 = new Account("02", "Natasha", 150);
        Account[] accounts2 = {b1, b2};
        System.out.println("Average balance is: " + average(accounts2));

        Measurable[] countries = {
            new Country("Indonesia", 1_904_569),
            new Country("Malaysia", 330_803),
            new Country("Singapore", 734)
        };
        System.out.println("Average area is: " + average(countries));

        Measurable y = new Account("01", "Aleksandra", 100);  // interface type, class
object
        System.out.println("y.getMeasure() = " + y.getMeasure());
    }
}
```

Compile it together with the interface (`javac Measurable.java MeasurableDemo.java`) and run `java MeasurableDemo`:

```
Average balance is: 125.0
Average balance is: 125.0
Average area is: 745368.6666666666
y.getMeasure() = 100.0
```

The results agree with **Listing 6.17**, but the program is built differently. Walk through the new pieces.

`class Account implements Measurable`. This is a promise, and the compiler holds the class to it: `Account` must contain a public `double getMeasure()`, which returns the balance; `Country` returns its area. A class may implement *several* interfaces, separated by commas — one of their main advantages.

What if a class breaks the promise? Delete the `getMeasure` method from `Country` and compile again:

```
MeasurableDemo.java:20: error: Country is not abstract and does not override abstract
method getMeasure() in Measurable
class Country implements Measurable {
^
1 error
```

The mistake is caught before the program ever runs.

`average(Measurable[] objects)`. The parameter's type is the interface, so inside the loop the only thing we may call on `obj` is `getMeasure()` — all the method needs. When `obj` refers to an `Account`, `Account`'s version runs; when it refers to a `Country`, `Country`'s. Which body runs is decided *at run time* by the class of the actual object: **dynamic dispatch**, the heart of *polymorphism*, which Chapter 7 develops. The arithmetic is the ordinary mean:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{6.4}$$

where x_i is the measure of the i-th object and n the number of objects. Equation (6.4) needs nothing but the n numbers, which is exactly why the interface needs nothing but `getMeasure`. For the three accounts, $\bar{x} = (100 + 150 + 125)/3 = 375/3 = 125$.

Two ways to call it. The program passes both a `Measurable[]` filled with accounts and a plain `Account[]`; Java accepts the second because every `Account` *is a* `Measurable`.

No objects of an interface. A variable may have an interface type — `Measurable y` — and refer to an object of an implementing class, but you can never create an object *of the interface itself*:

```
NewIface.java:4: error: Measurable is abstract; cannot be instantiated
        Measurable x = new Measurable();
                       ^
1 error
```

`new Measurable()` would be an object promising a `getMeasure` that has no body. An interface is a contract, not a thing.

> ✓ **KEY INSIGHT.** An interface splits a program into two sides that no longer need to know each other. The *caller* (`average`) knows only the interface. The *implementers* (`Account`, `Country`) know only the interface. Either side can change — a new implementer, a rewritten caller — without touching the other, as long as both keep the contract.

Python's version, **Listing 6.23**, uses a **protocol**, Python's way of writing an interface that is checked by *shape*. We return to protocols in Section 6.15; for now read the class `Measurable(Protocol)` as "any object with a method `get_measure()` returning a float".

Listing 6.23 `measurable_demo.py`: the `Measurable` protocol; notice that `Account` and `Country` never mention it.

```
"""One average function for everything that has a measure, using a Protocol."""
from collections.abc import Sequence
from typing import Protocol, runtime_checkable


@runtime_checkable
class Measurable(Protocol):
    """Anything that can be reduced to one number for averaging."""

    def get_measure(self) -> float:
        ...


class Account:            # note: no "Measurable" anywhere in this class
    def __init__(self, id, owner, balance):
        self.id = id
        self.owner = owner
        self.balance = balance

    def get_measure(self) -> float:
        return self.balance


class Country:
    def __init__(self, name, area):
        self.name = name
        self.area = area

    def get_measure(self) -> float:
        return self.area


def average(objects: Sequence[Measurable]) -> float:
    """Average measure of the objects, or 0 for an empty list."""
    if len(objects) == 0:
        return 0
    total = 0.0
    for obj in objects:
        total = total + obj.get_measure()
    return total / len(objects)
```

Listing 6.23 (Continued) *measurable_demo.py: the Measurable protocol; notice that Account and Country never mention it.*

```
if __name__ == "__main__":
    accounts = [Account("01", "Aleksandra", 100),
                Account("02", "Natasha", 150),
                Account("03", "Sergei", 125)]
    print("Average balance is:", average(accounts))
    countries = [Country("Indonesia", 1_904_569),
                 Country("Malaysia", 330_803),
                 Country("Singapore", 734)]
    print("Average area is:", average(countries))
    print("Account is Measurable:", isinstance(accounts[0], Measurable))
    print("a str is Measurable  :", isinstance("hello", Measurable))
```

```
Average balance is: 125.0
Average area is: 745368.6666666666
Account is Measurable: True
a str is Measurable  : False
```

The `...` in the protocol's method is Python's way of writing "no body here". `Account` and `Country` do not say `Measurable` anywhere, yet `isinstance(accounts[0], Measurable)` answers `True`: the

object has a `get_measure` method, so it fits. A string has no such method, so it does not. The annotation `Sequence[Measurable]` on `average` means "a list, tuple or other sequence of `Measurable` objects"; Python itself ignores annotations when running, but tools read them, as Section 6.15 shows.

6.12.1 Why interfaces matter: a plug-in story

Interfaces are not only a way to save a few duplicated lines. Picture a company whose image-editing filters are trade secrets, yet which wants other people to write *plug-ins* for its editor. It publishes an interface, say `interface ImageTool { String name(); Image apply(Image input); }`, with its `javadoc` pages. Anybody may write a class that implements `ImageTool` and hand in the compiled class; the editor puts `name()` into its menu and calls `apply` when the user clicks. Neither side ever sees the other's source: the interface is the whole of what they share. Browser extensions, editor plug-ins and printer drivers all follow this shape.

6.13 Swapping a Representation Behind an Interface

Here is a smaller version of the same story, one you can run. A program must keep a table of integer keys — a tiny "database" — with two operations: add a key, and search for one. There are two reasonable ways to store the keys. `LinearDB` keeps them in an unsorted list and searches by looking at each key in turn. `TreeDB` keeps them in a *binary search tree*: each key sits in a node, smaller keys go to the left, larger keys to the right, so a search follows one path from the top instead of looking at everything. Which is better depends on the data, so the client program wants to decide when it starts.

6.13.1 Without an interface

Listing 6.24 is the client written without an interface. The two classes are in package `dbrep`, so the file sits in a folder `dbrep`; compile it from the folder *above* with `javac dbrep/DBClient.java` and run it with `java dbrep.DBClient`.

Listing 6.24 `dbrep/DBClient.java`: a client that keeps one variable per representation and tests a flag at every operation.

```
package dbrep;

import java.util.ArrayList;
import java.util.List;

/** Keeps the keys in an unsorted list; a search looks at every key. */
class LinearDB {
    private final List<Integer> keys = new ArrayList<>();

    void addKey(int k) {
        System.out.println("LinearDB: adding key " + k);
        keys.add(k);
    }

    boolean search(int k) {
        System.out.println("LinearDB: searching for key " + k);
        for (int key : keys) {
            if (key == k) {
                return true;
            }
        }
        return false;
    }
}

/** Keeps the keys in a binary search tree; a search follows one path down. */
class TreeDB {
    private static class Node {
        final int key;
        Node left;
        Node right;

        Node(int key) {
            this.key = key;
        }
    }

    private Node root;
```

Listing 6.24 (Continued) *dbrep/DBClient.java: a client that keeps one variable per representation and tests a flag at every operation.*

```
    void addKey(int k) {
        System.out.println("TreeDB: adding key " + k);
        Node fresh = new Node(k);
        if (root == null) {
            root = fresh;
            return;
        }
        Node node = root;
        while (true) {
            if (k == node.key) {
                return;                          // already present
            } else if (k < node.key) {
                if (node.left == null) {
                    node.left = fresh;
                    return;
                }
                node = node.left;
            } else {
                if (node.right == null) {
                    node.right = fresh;
                    return;
                }
                node = node.right;
            }
        }
    }

    boolean search(int k) {
        System.out.println("TreeDB: searching for key " + k);
        Node node = root;
        while (node != null) {
            if (k == node.key) {
                return true;
            }
            node = (k < node.key) ? node.left : node.right;
        }
        return false;
    }
}
```

Listing 6.24 (Continued) *dbrep/DBClient.java: a client that keeps one variable per representation and tests a flag at every operation.*

```
/**
 * A client that must test, at every operation, which representation it chose.
 */
public class DBClient {
    static LinearDB ldb;
    static TreeDB tdb;
    static boolean useLinearDB = true;

    public static void main(String[] args) {
        // Decide which representation to use (the default is the tree)
        String decision = (args.length > 0) ? args[0] : "Tree Rep";
        if (decision.equals("Linear Rep")) {
            ldb = new LinearDB();
        } else {                                    // otherwise, the tree is used
            useLinearDB = false;
            tdb = new TreeDB();
        }
        // Add three keys
        int[] keys = {13, 9, 21};
        for (int k : keys) {
            if (useLinearDB) {
                ldb.addKey(k);
            } else {
                tdb.addKey(k);
            }
        }
        // Search for two keys
        int k2 = 9;
        boolean result = useLinearDB ? ldb.search(k2) : tdb.search(k2);
        System.out.println("found " + k2 + ": " + result);
        int k3 = 4;
        result = useLinearDB ? ldb.search(k3) : tdb.search(k3);
        System.out.println("found " + k3 + ": " + result);
    }
}
```

By default it chooses the tree:

```
TreeDB: adding key 13
TreeDB: adding key 9
TreeDB: adding key 21
TreeDB: searching for key 9
found 9: true
TreeDB: searching for key 4
found 4: false
```

and with the argument `"Linear Rep"` (`java dbrep.DBClient "Linear Rep"`) it chooses the list:

```
LinearDB: adding key 13
LinearDB: adding key 9
LinearDB: adding key 21
LinearDB: searching for key 9
found 9: true
LinearDB: searching for key 4
found 4: false
```

(How `TreeDB` works inside is a subject for a data-structures course; all that matters here is that it offers the same two operations as `LinearDB`.)

The program works, and it is already unpleasant. `main` keeps *two* variables, `ldb` and `tdb`, of which one is always `null`, and a flag, `useLinearDB`, that says which one is real. Every database operation is wrapped in a test of the flag. A real database has twenty operations, and the day somebody

adds a third representation, every one of those places must grow a third branch — and one of them will be forgotten. **Listing 6.25** is the same design in Python, with the same smell.

Listing 6.25 `db_client_v1.py`: the same flag-testing client in Python.

```
"""A client that must test, at every operation, which representation it chose."""
import sys


class LinearDB:
    """Keeps the keys in an unsorted list; a search looks at every key."""

    def __init__(self):
        self._keys = []

    def add_key(self, k):
        print("LinearDB: adding key", k)
        self._keys.append(k)

    def search(self, k):
        print("LinearDB: searching for key", k)
        for key in self._keys:
            if key == k:
                return True
        return False


class _Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


class TreeDB:
    """Keeps the keys in a binary search tree; a search follows one path down."""

    def __init__(self):
        self._root = None

    def add_key(self, k):
        print("TreeDB: adding key", k)
        fresh = _Node(k)
        if self._root is None:
            self._root = fresh
            return
        node = self._root
        while True:
            if k == node.key:
                return                          # already present
            elif k < node.key:
```

Listing 6.25 (Continued) *db_client_v1.py: the same flag-testing client in Python.*

```
                if node.left is None:
                    node.left = fresh
                    return
                node = node.left
            else:
                if node.right is None:
                    node.right = fresh
                    return
                node = node.right

    def search(self, k):
        print("TreeDB: searching for key", k)
        node = self._root
        while node is not None:
            if k == node.key:
                return True
            node = node.left if k < node.key else node.right
        return False


ldb = None
tdb = None
use_linear_db = True

# Decide which representation to use (the default is the tree)
decision = sys.argv[1] if len(sys.argv) > 1 else "Tree Rep"
if decision == "Linear Rep":
    ldb = LinearDB()
else:                                          # otherwise, the tree is used
    use_linear_db = False
    tdb = TreeDB()
# Add three keys
for k in [13, 9, 21]:
    if use_linear_db:
        ldb.add_key(k)
    else:
        tdb.add_key(k)
# Search for two keys
k2 = 9
result = ldb.search(k2) if use_linear_db else tdb.search(k2)
print(f"found {k2}: {str(result).lower()}")
k3 = 4
result = ldb.search(k3) if use_linear_db else tdb.search(k3)
print(f"found {k3}: {str(result).lower()}")
```

```
TreeDB: adding key 13
TreeDB: adding key 9
TreeDB: adding key 21
TreeDB: searching for key 9
found 9: true
TreeDB: searching for key 4
found 4: false
```

6.13.2 With an interface

The operations both representations offer are `addKey` and `search`, so we write them down as an interface, `DBops`, and make both classes implement it: add `implements DBops` to each header and declare the two methods `public` (a class may not implement an interface method with less visibility). Nothing else in the two classes changes.

The client changes a great deal. **Listing 6.26** is the result.

Listing 6.26 `dbiface/DBClient.java`: one variable of the interface type, no flag, and no tests.

```
package dbiface;

import java.util.ArrayList;
import java.util.List;

/** The operations every representation of the table must offer. */
interface DBops {
    void addKey(int k);

    boolean search(int k);

    /** Adds several keys; written once here, inherited by every implementation. */
    default void addSeveral(int[] keys) {
        for (int k : keys) {
            addKey(k);
        }
    }

    /** Tells whether at least one of the keys is present. */
    default boolean findOneOf(int[] keys) {
        for (int k : keys) {
            if (search(k)) {
                return true;
            }
        }
        return false;
    }
}

/** Keeps the keys in an unsorted list; a search looks at every key. */
class LinearDB implements DBops {
    private final List<Integer> keys = new ArrayList<>();

    @Override
    public void addKey(int k) {
        System.out.println("LinearDB: adding key " + k);
        keys.add(k);
    }
```

Listing 6.26 (Continued) *dbiface/DBClient.java: one variable of the interface type, no flag, and no tests.*

```
    @Override
    public boolean search(int k) {
        System.out.println("LinearDB: searching for key " + k);
        for (int key : keys) {
            if (key == k) {
                return true;
            }
        }
        return false;
    }
}

/** Keeps the keys in a binary search tree; a search follows one path down. */
class TreeDB implements DBops {
    private static class Node {
        final int key;
        Node left;
        Node right;

        Node(int key) {
            this.key = key;
        }
    }

    private Node root;

    @Override
    public void addKey(int k) {
        System.out.println("TreeDB: adding key " + k);
        Node fresh = new Node(k);
        if (root == null) {
            root = fresh;
            return;
        }
        Node node = root;
        while (true) {
            if (k == node.key) {
                return;                          // already present
            } else if (k < node.key) {
                if (node.left == null) {
                    node.left = fresh;
                    return;
                }
                node = node.left;
            } else {
                if (node.right == null) {
```

Listing 6.26 (Continued) *dbiface/DBClient.java: one variable of the interface type, no flag, and no tests.*

```
                    node.right = fresh;
                    return;
                }
                node = node.right;
            }
        }
    }

    @Override
    public boolean search(int k) {
        System.out.println("TreeDB: searching for key " + k);
        Node node = root;
        while (node != null) {
            if (k == node.key) {
                return true;
            }
            node = (k < node.key) ? node.left : node.right;
        }
        return false;
    }
}

/**
 * The same client, written against the interface: no tests, no second variable.
 */
public class DBClient {
    static DBops db;

    public static void main(String[] args) {
        // Decide which representation to use (the default is the tree)
        String decision = (args.length > 0) ? args[0] : "Tree Rep";
        if (decision.equals("Linear Rep")) {
            db = new LinearDB();
        } else {
            db = new TreeDB();
        }
        db.addSeveral(new int[] {13, 9, 21});
        System.out.println("found 9: " + db.search(9));
        System.out.println("found 4: " + db.search(4));
        System.out.println("found one of {4, 21}: " + db.findOneOf(new int[] {4, 21}));
    }
}
```

```
TreeDB: adding key 13
TreeDB: adding key 9
TreeDB: adding key 21
TreeDB: searching for key 9
found 9: true
TreeDB: searching for key 4
found 4: false
TreeDB: searching for key 4
TreeDB: searching for key 21
found one of {4, 21}: true
```

`main` now has *one* variable, `db`, of the interface type `DBops`. The decision between `LinearDB` and `TreeDB` is made once, at the top, and never mentioned again; every operation is an ordinary call on `db`, and dynamic dispatch runs the right code. A third representation would be one new class and one more line in the decision.

Figure 6.5 draws the new structure in UML, the standard diagram language for classes. A box with «interface» above its name is an interface; a dashed line with a hollow triangle means "implements"; a plain arrow means "uses".

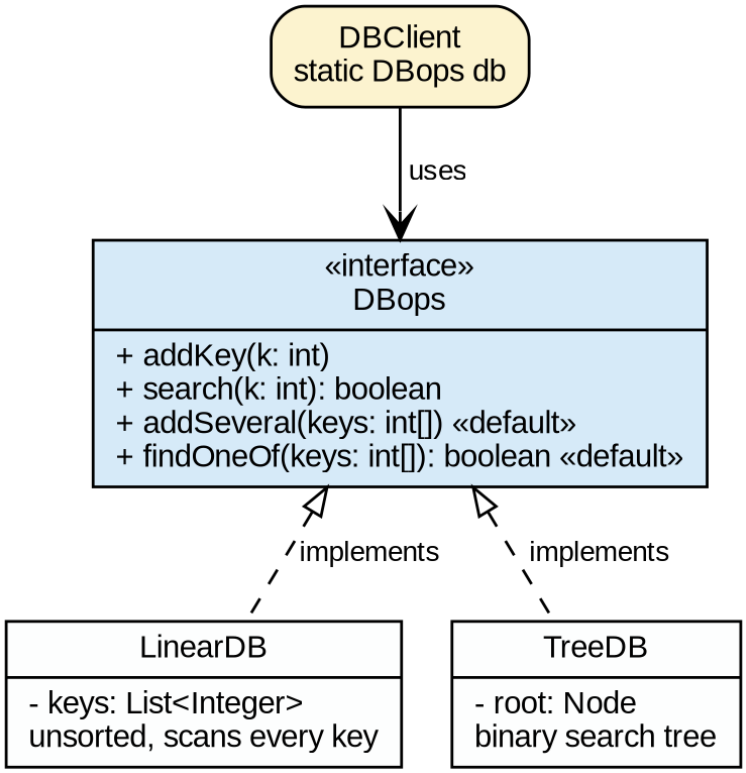


Figure 6.5 The client depends only on the interface `DBops`; `LinearDB` and `TreeDB` implement it and can be exchanged without changing the client.

6.13.3 Type hierarchies: subtypes and supertypes

Once `LinearDB` and `TreeDB` implement `DBops`, the three types form a small *hierarchy*: `DBops` is a **supertype** of each class, and each class is a **subtype** of `DBops`. Every object of a subtype can be regarded as an object of the supertype. In an assignment such as `db = new TreeDB();`, or when an `Account[]` is passed where a `Measurable[]` is expected, Java converts from the subtype to the supertype *automatically*: nothing can go wrong, because a `TreeDB` offers everything a `DBops` promises. Section 6.14.1 looks at the opposite direction, which is not automatic.

6.13.4 Sharing code: default methods

Now suppose both representations need two more operations: `addSeveral(int[] keys)`, which adds a whole array of keys, and `findOneOf(int[] keys)`, which tells whether at least one of several keys is present. Both are easy to write in terms of `addKey` and `search` — a loop over the array — and the loop would be identical in `LinearDB` and `TreeDB`. Should we copy it into both classes?

Since Java 8 an interface may contain **default methods**: methods with a body, marked `default`, which every implementing class inherits unless it provides its own. In **Listing 6.26**, `addSeveral` and `findOneOf` are default methods, written once from the two abstract methods and inherited by both classes; the last lines of the output show `findOneOf` calling `search` for `4` and then `21`. An interface may also contain `static` methods, which belong to the interface itself; Section 6.16 uses one.

Default methods can use only the interface's own methods, because an interface has no fields: it describes behavior, not state. When implementations must also share fields, an abstract class is the tool (Chapter 7); **Table 6.6** below weighs the two.

The Python version uses an **abstract base class**, which can hold shared code directly. **Listing 6.27** shows it.

Listing 6.27 `db_client_v2.py`: the interface as an abstract base class, with two shared methods.

```
"""The same client, written against an abstract base class."""
import sys
from abc import ABC, abstractmethod


class DBops(ABC):
    """The operations every representation of the table must offer."""

    @abstractmethod
    def add_key(self, k):
        ...

    @abstractmethod
    def search(self, k):
        ...

    def add_several(self, keys):
        """Add several keys; written once here, inherited by every subclass."""
        for k in keys:
            self.add_key(k)

    def find_one_of(self, keys):
        """Tell whether at least one of the keys is present."""
        for k in keys:
            if self.search(k):
                return True
        return False


class LinearDB(DBops):
    """Keeps the keys in an unsorted list; a search looks at every key."""

    def __init__(self):
        self._keys = []

    def add_key(self, k):
        print("LinearDB: adding key", k)
        self._keys.append(k)

    def search(self, k):
        print("LinearDB: searching for key", k)
        for key in self._keys:
            if key == k:
                return True
        return False
```

Listing 6.27 (Continued) *db_client_v2.py: the interface as an abstract base class, with two shared methods.*

```
class _Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


class TreeDB(DBops):
    """Keeps the keys in a binary search tree; a search follows one path down."""

    def __init__(self):
        self._root = None

    def add_key(self, k):
        print("TreeDB: adding key", k)
        fresh = _Node(k)
        if self._root is None:
            self._root = fresh
            return
        node = self._root
        while True:
            if k == node.key:
                return                          # already present
            elif k < node.key:
                if node.left is None:
                    node.left = fresh
                    return
                node = node.left
            else:
                if node.right is None:
                    node.right = fresh
                    return
                node = node.right

    def search(self, k):
        print("TreeDB: searching for key", k)
        node = self._root
        while node is not None:
            if k == node.key:
                return True
            node = node.left if k < node.key else node.right
        return False
```

Listing 6.27 (Continued) *db_client_v2.py: the interface as an abstract base class, with two shared methods.*

```
# Decide which representation to use (the default is the tree)
decision = sys.argv[1] if len(sys.argv) > 1 else "Tree Rep"
db: DBops = LinearDB() if decision == "Linear Rep" else TreeDB()
db.add_several([13, 9, 21])
print("found 9:", str(db.search(9)).lower())
print("found 4:", str(db.search(4)).lower())
print("found one of {4, 21}:", str(db.find_one_of([4, 21])).lower())
try:
    DBops()
except TypeError as e:
    print("DBops() fails:", e)
```

```
TreeDB: adding key 13
TreeDB: adding key 9
TreeDB: adding key 21
TreeDB: searching for key 9
found 9: true
TreeDB: searching for key 4
found 4: false
TreeDB: searching for key 4
TreeDB: searching for key 21
found one of {4, 21}: true
DBops() fails: Can't instantiate abstract class DBops with abstract methods add_key, search
```

`DBops` inherits from `ABC`, and its two operations carry the decorator `@abstractmethod`. The shared methods `add_several` and `find_one_of` are ordinary methods written in the base class; `LinearDB(DBops)` and `TreeDB(DBops)` inherit them. The last line proves that the abstract class cannot be instantiated: `DBops()` raises a `TypeError` that names the missing methods, just as Java refused `new Measurable()`. (Python 3.12 and later word it "without an implementation for abstract methods".)

6.14 Interfaces You Already Use

The Java library is full of interfaces, and you have been using some of them without knowing. **Table 6.5** lists those a beginner meets first, with their Python counterparts. The packages `java.lang` (loaded automatically) and `java.util` (collections) hold most of them; the complete lists are in the API documentation of each package.

Table 6.5 Interfaces from the Java library that appear early, and what plays the same part in Python.

Interface	Package	Promises	Python counterpart
`Comparable<T>`	`java.lang`	a natural order: `compareTo(T other)`	`__lt__` (and `functools.total_ordering`)
`Comparator<T>`	`java.util`	an ordering chosen from outside: `compare(a, b)`	a `key=` function for `sorted`
`Iterable<T>`	`java.lang`	usable in a for-each loop: `iterator()`	`__iter__`, `collections.abc.Iterable`
`CharSequence`	`java.lang`	a readable sequence of `char`	`str`, `collections.abc.Sequence`
`AutoCloseable`	`java.lang`	releases resources: `close()`, for try-with-resources	`__enter__`/`__exit__`, the `with` statement
`Runnable`	`java.lang`	a task a thread can run: `run()`	any callable, e.g. a function
`Collection<E>`, `List<E>`, `Set<E>`, `Map<K,V>`	`java.util`	the collection operations	`collections.abc.Collection`, `Sequence`, `Set`, `Mapping`
`ActionListener`	`java.awt.event`	reacts to a button: `actionPerformed(e)`	a callback function (Tkinter `command=`)

You wrote `List<Order> orders = new ArrayList<>();` in Chapter 5 — `List` is an interface and `ArrayList` one of its implementations, which is why the same code works with a `LinkedList`. When a for-each loop runs over a list, it relies on `Iterable`. When try-with-resources closed your file in Chapter 5, it relied on `AutoCloseable`. The rest of this section looks at two more closely: `Comparable`, which sorting depends on, and `ActionListener`, which every button depends on.

6.14.1 Comparable, and casting down a hierarchy

Most collections can be sorted, and sorting is a long sequence of comparisons: "should this element come before that one?" For `Arrays.sort` to sort objects of *your* class, it must be able to ask that question of them, and it asks through the interface `Comparable`. Its single method, `int compareTo(T other)`, answers with the sign of an integer:

$$\operatorname{sign}(\texttt{a.compareTo(b)}) = \begin{cases} -1 & \text{if } a \text{ comes before } b \\ 0 & \text{if they are equal in order} \\ +1 & \text{if } a \text{ comes after } b \end{cases} \tag{6.5}$$

Only the *sign* is promised — Equation (6.5) speaks of the sign, not the value. `"Ace".compareTo("Zonk")` is −25 (the difference between the codes of `'A'` and `'Z'`), not −1. Keep that in mind; it is the whole of the story below.

Suppose each name has a last name and a first name, and names should sort by last name, then by first name. **Listing 6.28** contains two versions of the class side by side. `SwitchName` is written in an older style: it implements the *raw* `Comparable` — without the `<T>` — whose method receives a plain `Object`, and it decides with a `switch` on the result of comparing last names. `Name` is the correct modern version.

Listing 6.28 `NameSortDemo.java`: a correct `compareTo` and one that trusts `String.compareTo` to return only −1, 0 or 1.

```
import java.util.Arrays;

/**
 * A name compared with a switch that expects compareTo to return only -1, 0 or 1.
 * It uses the raw (pre-generics) Comparable, so it must cast its argument.
 */
@SuppressWarnings("rawtypes")
class SwitchName implements Comparable {
    private final String lastName;
    private final String firstName;

    SwitchName(String lastName, String firstName) {
        this.lastName = lastName;
        this.firstName = firstName;
    }

    @Override
    public int compareTo(Object other) {
        SwitchName o = (SwitchName) other;           // down-cast: Object -> SwitchName
        int result = Integer.MIN_VALUE;
        switch (lastName.compareTo(o.lastName)) {
            case -1:
                result = -1;
                break;
            case 1:
                result = 1;
                break;
            case 0:
                result = firstName.compareTo(o.firstName);
                break;
        }
        return result;                               // MIN_VALUE for every other
difference!
    }

    @Override
    public String toString() {
        return "[" + lastName + ", " + firstName + "]";
    }
}
```

Listing 6.28 (Continued) *NameSortDemo.java: a correct compareTo and one that trusts String.compareTo to return only -1, 0 or 1.*

```
/**
 * A name ordered by last name, then by first name: the correct version.
 */
class Name implements Comparable<Name> {
    private final String lastName;
    private final String firstName;

    Name(String lastName, String firstName) {
        this.lastName = lastName;
        this.firstName = firstName;
    }

    @Override
    public int compareTo(Name other) {               // no cast needed: other is already a
Name
        int byLast = lastName.compareTo(other.lastName);
        if (byLast != 0) {
            return byLast;                           // any negative or positive number
will do
        }
        return firstName.compareTo(other.firstName);
    }

    @Override
    public String toString() {
        return "[" + lastName + ", " + firstName + "]";
    }
}

/**
 * Sorts the same names with both versions.
 */
public class NameSortDemo {
    public static void main(String[] args) {
        Name[] names = {
            new Name("Zonk", "Godot"),
            new Name("Bogdanov", "Alexei"),
            new Name("Ace", "Portgas D."),
            new Name("Bogdanov", "Aleksandra")
        };
        System.out.println("Original order:");
        for (Name n : names) {
            System.out.println(n);
        }
        Arrays.sort(names);                          // uses Name.compareTo
        System.out.println("\nAfter sorting:");
        for (Name n : names) {
            System.out.println(n);
```

Listing 6.28 (Continued) *NameSortDemo.java: a correct compareTo and one that trusts String.compareTo to return only -1, 0 or 1.*

```
        }

        SwitchName[] risky = {
            new SwitchName("Bogdanov", "Alexei"),
            new SwitchName("Zonk", "Godot"),
            new SwitchName("Ace", "Portgas D."),
            new SwitchName("Bogdanov", "Aleksandra")
        };
        Arrays.sort(risky);                          // uses SwitchName.compareTo
        System.out.println("\nSwitch version after sorting:");
        for (SwitchName n : risky) {
            System.out.println(n);
        }
    }
}
```

```
Original order:
[Zonk, Godot]
[Bogdanov, Alexei]
[Ace, Portgas D.]
[Bogdanov, Aleksandra]

After sorting:
[Ace, Portgas D.]
[Bogdanov, Aleksandra]
[Bogdanov, Alexei]
[Zonk, Godot]

Switch version after sorting:
[Ace, Portgas D.]
[Bogdanov, Aleksandra]
[Zonk, Godot]
[Bogdanov, Alexei]
```

`Name` sorts perfectly, but `SwitchName` puts Zonk *between* the two Bogdanovs. Its `switch` has cases only for −1, 1 and 0. Comparing “Bogdanov” with “Zonk” gives −24, which matches no case, so `result` keeps its initial value `Integer.MIN_VALUE`, meaning “comes before” — correct by luck. Comparing “Zonk” with “Bogdanov” gives +24, *also* no case, *also* `MIN_VALUE`. The method now claims that each name comes before the other, and the sorted order depends on the starting order: with three names the bug hides, with four it shows — which is why Section 6.8 insisted on choosing test inputs deliberately.

`Name` avoids the trap by never looking at the *value* of `compareTo`, only at whether it is zero: if the last names differ, their comparison is the answer, whatever number it is; otherwise the first names decide.

The raw version also shows a **cast**. Its `compareTo` receives an `Object`, and must write `SwitchName o = (SwitchName) other;` before it can read `o.lastName`. **Figure 6.6** shows why the cast is needed.

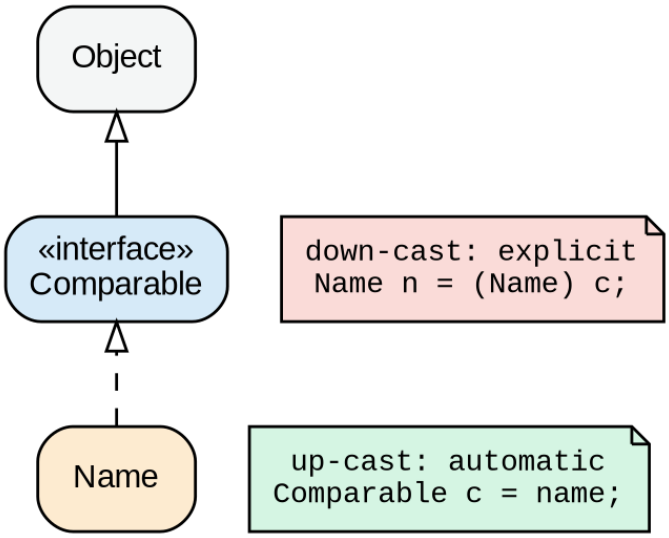


Figure 6.6 Moving up the type hierarchy (from `Name` to `Comparable` or `Object`) is automatic; moving down needs an explicit cast, which is checked when the program runs.

Converting from a subtype to a supertype — **up-casting** — is automatic. Converting down — **down-casting** — needs an explicit cast, `(Name) c`, because it might be wrong: a `Comparable` variable could hold a `String`. The virtual machine checks the cast at run time and throws a `ClassCastException` if it fails. The generic `Comparable<Name>` removes the cast altogether, because its `compareTo` receives a `Name`; prefer it always (Chapter 9 explains generics).

Python sorts by asking `a < b`, which calls `a.__lt__(b)`. **Listing 6.29** gives `Name` a `__lt__` that compares the *tuples* `(last_name, first_name)`, since Python compares tuples item by item; `@functools.total_ordering` fills in `<=`, `>` and `>=`. The file also ports the switch version through `functools.cmp_to_key`, and shows the most Pythonic way: a `key=` function that says *what* to sort by.

Listing 6.29 `name_sort.py`: ordering by a tuple key, the same switch bug, and sorting with a `key` function.

```
"""Ordering names: the Pythonic way, and a port of the switch-based compareTo."""
import sys
from functools import cmp_to_key, total_ordering


def java_compare(a, b):
    """Compare two strings the way Java's String.compareTo does."""
    for x, y in zip(a, b):
        if x != y:
            return ord(x) - ord(y)          # the difference of the first unequal
characters
    return len(a) - len(b)


@total_ordering                             # builds <=, >, >= from __eq__ and __lt__
class Name:
    """A name ordered by last name, then by first name."""

    def __init__(self, last_name, first_name):
        self.last_name = last_name
        self.first_name = first_name

    def _key(self):
        return (self.last_name, self.first_name)

    def __eq__(self, other):
        if not isinstance(other, Name):
            return NotImplemented
        return self._key() == other._key()

    def __lt__(self, other):
        if not isinstance(other, Name):
            return NotImplemented
        return self._key() < other._key()   # tuples compare item by item

    def __hash__(self):
        return hash(self._key())

    def __repr__(self):
        return f"[{self.last_name}, {self.first_name}]"
```

Listing 6.29 (Continued) *name_sort.py: ordering by a tuple key, the same switch bug, and sorting with a key function.*

```
def switch_compare(a, b):
    """The switch version: correct only when the comparison gives -1, 0 or 1."""
    result = -sys.maxsize - 1
    match java_compare(a.last_name, b.last_name):
        case -1:
            result = -1
        case 1:
            result = 1
        case 0:
            result = java_compare(a.first_name, b.first_name)
    return result                           # a huge negative number for every other
difference!


names = [Name("Zonk", "Godot"), Name("Bogdanov", "Alexei"),
         Name("Ace", "Portgas D."), Name("Bogdanov", "Aleksandra")]
print("Original order:")
for n in names:
    print(n)
print("\nAfter sorting:")
for n in sorted(names):                     # uses Name.__lt__
    print(n)

risky = [Name("Bogdanov", "Alexei"), Name("Zonk", "Godot"),
         Name("Ace", "Portgas D."), Name("Bogdanov", "Aleksandra")]
print("\nSwitch version after sorting:")
for n in sorted(risky, key=cmp_to_key(switch_compare)):
    print(n)

print("\nBy first name, using a key function:")
for n in sorted(names, key=lambda n: n.first_name):
    print(n)
```

```
Original order:
[Zonk, Godot]
[Bogdanov, Alexei]
[Ace, Portgas D.]
[Bogdanov, Aleksandra]

After sorting:
[Ace, Portgas D.]
[Bogdanov, Aleksandra]
[Bogdanov, Alexei]
[Zonk, Godot]

Switch version after sorting:
[Ace, Portgas D.]
[Bogdanov, Aleksandra]
[Zonk, Godot]
[Bogdanov, Alexei]

By first name, using a key function:
[Bogdanov, Aleksandra]
[Bogdanov, Alexei]
[Zonk, Godot]
[Ace, Portgas D.]
```

The switch port fails in exactly the same way, which shows that the bug lives in the idea, not in the language.

6.14.2 Listeners: the idea behind ActionListener

A window button must tell other objects when it is clicked, yet the people who wrote the `JButton` class could not know what *your* objects would be. Java's answer is an interface in the package `java.awt.event`:

```
interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

A button keeps a list of listeners, offers `addActionListener` and `removeActionListener`, and when clicked calls `actionPerformed` on every listener in its list. Bouncing balls, swinging pendulums and rotating spirals can all listen to one *Stop* button, which needs to know nothing about them.

Chapter 10 builds real windows. **Listing 6.30** shows the same idea without a window, so that you can run it anywhere.

Listing 6.30 `StopButtonDemo.java`: a button that notifies every registered listener through an interface.

```
import java.util.ArrayList;
import java.util.List;

/** Anything that wants to hear when the Stop button is pressed. */
interface StopListener {
    void stopRequested(String reason);
}

/** A bouncing ball in an animation. */
class BouncingBall implements StopListener {
    @Override
    public void stopRequested(String reason) {
        System.out.println("  ball: stops bouncing (" + reason + ")");
    }
}

/** A swinging pendulum. */
class Pendulum implements StopListener {
    @Override
    public void stopRequested(String reason) {
        System.out.println("  pendulum: comes to rest (" + reason + ")");
    }
}

/** A rotating spiral. */
class RotatingSpiral implements StopListener {
    @Override
    public void stopRequested(String reason) {
        System.out.println("  spiral: stops rotating (" + reason + ")");
    }
}

/** A button that knows only the interface of the objects it notifies. */
class StopButton {
    private final List<StopListener> listeners = new ArrayList<>();

    void addStopListener(StopListener listener) {
        listeners.add(listener);
    }

    void removeStopListener(StopListener listener) {
        listeners.remove(listener);
    }
```

Listing 6.30 (Continued) *StopButtonDemo.java: a button that notifies every registered listener through an interface.*

```
    void click(String reason) {
        System.out.println("Stop pressed, " + listeners.size() + " listener(s):");
        for (StopListener listener : listeners) {
            listener.stopRequested(reason);
        }
    }
}

/**
 * The listener idea behind ActionListener, without a window.
 */
public class StopButtonDemo {
    public static void main(String[] args) {
        StopButton stop = new StopButton();
        BouncingBall ball = new BouncingBall();
        stop.addStopListener(ball);
        stop.addStopListener(new Pendulum());
        stop.addStopListener(new RotatingSpiral());
        stop.click("first click");
        stop.removeStopListener(ball);
        stop.click("second click");
    }
}
```

```
Stop pressed, 3 listener(s):
  ball: stops bouncing (first click)
  pendulum: comes to rest (first click)
  spiral: stops rotating (first click)
Stop pressed, 2 listener(s):
  pendulum: comes to rest (second click)
  spiral: stops rotating (second click)
```

`StopButton` contains the word `StopListener` and nothing else about its listeners; the three animated objects contain `implements StopListener` and nothing else about the button. The removed ball no longer hears the second click. **Listing 6.31** is the Python version; in Python a plain function could serve as a listener too, which is what Tkinter's `command=` does.

Listing 6.31 `stop_button_demo.py`: the same listener pattern with a Python protocol.

```
"""The listener idea behind ActionListener, without a window."""
from typing import Protocol


class StopListener(Protocol):
    """Anything that wants to hear when the Stop button is pressed."""

    def stop_requested(self, reason: str) -> None:
        ...


class BouncingBall:
    def stop_requested(self, reason):
        print(f"  ball: stops bouncing ({reason})")


class Pendulum:
    def stop_requested(self, reason):
        print(f"  pendulum: comes to rest ({reason})")


class RotatingSpiral:
    def stop_requested(self, reason):
        print(f"  spiral: stops rotating ({reason})")


class StopButton:
    """A button that knows only the protocol of the objects it notifies."""

    def __init__(self):
        self._listeners: list[StopListener] = []

    def add_stop_listener(self, listener: StopListener):
        self._listeners.append(listener)

    def remove_stop_listener(self, listener: StopListener):
        self._listeners.remove(listener)

    def click(self, reason):
        print(f"Stop pressed, {len(self._listeners)} listener(s):")
        for listener in self._listeners:
            listener.stop_requested(reason)
```

Listing 6.31 (Continued) *stop_button_demo.py: the same listener pattern with a Python protocol.*

```
stop = StopButton()
ball = BouncingBall()
stop.add_stop_listener(ball)
stop.add_stop_listener(Pendulum())
stop.add_stop_listener(RotatingSpiral())
stop.click("first click")
stop.remove_stop_listener(ball)
stop.click("second click")
```

```
Stop pressed, 3 listener(s):
  ball: stops bouncing (first click)
  pendulum: comes to rest (first click)
  spiral: stops rotating (first click)
Stop pressed, 2 listener(s):
  pendulum: comes to rest (second click)
  spiral: stops rotating (second click)
```

6.15 Python's Interfaces: Abstract Base Classes and Protocols

Python has no `interface` keyword: it calls a method on whatever object it is given, and if the method exists, it runs. This is **duck typing** — "if it walks like a duck and quacks like a duck, it is a duck" — and its cost is that a missing method is discovered only when the call happens. Python therefore offers two ways of writing an interface down, both used in this chapter.

Abstract base classes (ABCs, from the module `abc` [13]) are the *nominal* way: a class belongs to the interface because it says so, by inheriting from it. `DBops(ABC)` in **Listing 6.27** and `PriceRule(ABC)` in Section 6.16 are ABCs. Python enforces them at the moment an object is created: a subclass that forgets an `@abstractmethod` cannot be instantiated, and the `TypeError` names the missing method. ABCs can also carry shared code, as `add_several` showed.

Protocols (from `typing` [14]) are the *structural* way: a class belongs to the interface because it has the right methods, whether or not it has ever heard of the protocol — duck typing, written down. `Measurable` and `StopListener` are protocols. Python itself does *not* check a protocol when the program runs (except for `isinstance`, if the protocol is marked `@runtime_checkable`, and even then it checks only that the method names exist). The checking is done beforehand by a **static type checker** such as mypy, which reads the annotations. Given a file that passes a `Dog` (no `get_measure`) to `average`, mypy reports, without running anything:

```
bad_measure.py:9: error: List item 0 has incompatible type "Dog"; expected "Measurable"
[list-item]
Found 1 error in 1 file (checked 1 source file)
```

while running the same file gets as far as the loop and then fails with `AttributeError: 'Dog' object has no attribute 'get_measure'`. **Table 6.6** compares the four ways of stating a contract.

Table 6.6 Four ways to state "every class of this kind must provide these methods".

	Java interface	Java abstract class (Ch. 7)	Python ABC	Python Protocol
a class joins by	`implements`	`extends`	inheriting from it	having the methods
how many per class	any number	exactly one	any number	any number
may hold shared code	default and static methods	yes	yes	yes, but classes rarely inherit it
may hold fields (state)	constants only	yes	yes	no
a missing method is caught	at compile time	at compile time	when an object is created	by a type checker, before running

Use an ABC when you own all the implementations and want a loud failure the moment one is incomplete, or when they should share code. Use a Protocol when the implementations come from elsewhere or you would rather not force a common base class on anyone. Either is far better than an interface that exists only in a programmer's head.

6.16 Programming to an Interface: Pricing at Profitina Laundry

Everything in this chapter now comes together in one piece of working software. Profitina Laundry charges for its work in several ways, and the ways keep changing:

- **wash and fold** and **wash and iron** are charged per kilogram, every started kilogram counting, with a 3 kg minimum;
- **dry cleaning** is charged per piece;
- **shoe cleaning** has one fixed price;
- **express** service (ready the same day) costs 50% more, whatever the service;
- **members** get 10% off, whatever the service and whether or not it is express.

Next month there will be a rainy-season promotion; next year, a price for hotels. If the receipt code contains one `if` for each of these rules, every new rule means editing the receipt code, and every edit risks the rules already there. The design principle that prevents this is old and short [15]:

> ✓ **KEY INSIGHT.** *Program to an interface, not an implementation.* Code that needs a price should ask for "something that can price an order" — an interface — and never for a particular pricing class. Then new ways of pricing are new classes, and the code that uses prices never changes.

6.16.1 The order

First, the thing being priced. **Listing 6.32** is an `Order`: a ticket id, a customer's name, a weight, a number of pieces, and whether the customer is a member. The constructor refuses nonsense — a blank id, a negative quantity, or an order with neither weight nor pieces — so no pricing rule ever has to worry about it. Orders are identified by their ticket id, so `equals` and `hashCode` use the id alone, as Section 6.9 taught; `toString` gives the developer's view.

Listing 6.32 `Order.java`: a laundry order, validated on construction and identified by its ticket id.

```java
/**
 * One laundry order as written on the ticket at the counter.
 *
 * <p>Orders are identified by their ticket id: two {@code Order} objects
 * with the same id are the same order.</p>
 */
public final class Order {
    private final String id;
    private final String customer;
    private final double weightKg;
    private final int pieces;
    private final boolean member;

    /**
     * Creates an order.
     *
     * @param id       the ticket id, for example "A001"
     * @param customer the customer's name
     * @param weightKg the weight on the scale, in kilograms (0 if priced per piece)
     * @param pieces   the number of pieces (0 if priced by weight)
     * @param member   whether the customer holds a member card
     * @throws IllegalArgumentException if the id is blank, a quantity is negative,
     *                                  or the order has neither weight nor pieces
     */
    public Order(String id, String customer, double weightKg, int pieces, boolean member) {
        if (id == null || id.isBlank()) {
            throw new IllegalArgumentException("an order needs a ticket id");
        }
        if (weightKg < 0 || pieces < 0) {
            throw new IllegalArgumentException("quantities must not be negative");
        }
        if (weightKg == 0 && pieces == 0) {
            throw new IllegalArgumentException("an order needs a weight or some pieces");
        }
        this.id = id;
        this.customer = customer;
        this.weightKg = weightKg;
        this.pieces = pieces;
        this.member = member;
    }

    /** @return the ticket id */
    public String getId() {
        return id;
    }
```

Listing 6.32 (Continued) *Order.java: a laundry order, validated on construction and identified by its ticket id.*

```
    /** @return the customer's name */
    public String getCustomer() {
        return customer;
    }

    /** @return the weight in kilograms */
    public double getWeightKg() {
        return weightKg;
    }

    /** @return the number of pieces */
    public int getPieces() {
        return pieces;
    }

    /** @return whether the customer is a member */
    public boolean isMember() {
        return member;
    }

    @Override
    public boolean equals(Object other) {
        if (this == other) {
            return true;
        }
        if (!(other instanceof Order)) {
            return false;
        }
        return id.equals(((Order) other).id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }

    @Override
    public String toString() {
        return "Order[" + id + ", " + customer + ", " + weightKg + " kg, "
                + pieces + " pcs" + (member ? ", member" : "") + "]";
    }
}
```

The getters carry only an `@return` line. For a getter whose name says everything, that short form is accepted practice (although `javadoc` notes that the main description is missing); methods that do real work, like `priceOf` below, deserve the full treatment. **Listing 6.33** is the Python `Order`.

Listing 6.33 `order.py`: the same order in Python, with read-only properties.

```
"""One laundry order as written on the ticket at the counter."""


class Order:
    """One laundry order. Orders with the same ticket id are the same order."""

    def __init__(self, id, customer, weight_kg, pieces, member):
        """Create an order.

        Args:
            id: the ticket id, for example "A001".
            customer: the customer's name.
            weight_kg: the weight on the scale in kilograms (0 if priced per piece).
            pieces: the number of pieces (0 if priced by weight).
            member: whether the customer holds a member card.

        Raises:
            ValueError: if the id is blank, a quantity is negative,
                or the order has neither weight nor pieces.
        """
        if not id or not id.strip():
            raise ValueError("an order needs a ticket id")
        if weight_kg < 0 or pieces < 0:
            raise ValueError("quantities must not be negative")
        if weight_kg == 0 and pieces == 0:
            raise ValueError("an order needs a weight or some pieces")
        self._id = id
        self._customer = customer
        self._weight_kg = weight_kg
        self._pieces = pieces
        self._member = member

    @property
    def id(self):
        return self._id

    @property
    def customer(self):
        return self._customer

    @property
    def weight_kg(self):
        return self._weight_kg
```

Listing 6.33 (Continued) *order.py: the same order in Python, with read-only properties.*

```
    @property
    def pieces(self):
        return self._pieces

    @property
    def member(self):
        return self._member

    def __eq__(self, other):
        if not isinstance(other, Order):
            return NotImplemented
        return self._id == other._id

    def __hash__(self):
        return hash(self._id)

    def __repr__(self):
        extra = ", member" if self._member else ""
        return (f"Order[{self._id}, {self._customer}, {self._weight_kg} kg, "
                f"{self._pieces} pcs{extra}]")
```

6.16.2 The interface

Listing 6.34 is the contract every price rule must keep.

Listing 6.34 `PriceRule.java`: the interface every price rule implements, with a static helper that formats rupiah.

```java
import java.util.Locale;

/**
 * A rule that turns a laundry order into a price.
 *
 * <p>Every way Profitina Laundry charges for its work is a class that
 * implements this interface. Code that only needs a price asks for a
 * {@code PriceRule} and never learns which rule it received.</p>
 */
public interface PriceRule {
    /**
     * Computes the price of an order.
     *
     * @param order the order to price
     * @return the price in whole rupiah, never negative
     * @throws IllegalArgumentException if this rule cannot price the order
     */
    long priceOf(Order order);

    /**
     * Names the rule for the receipt.
     *
     * @return a short, human-readable name such as "wash & fold"
     */
    String name();

    /**
     * Formats an amount the Indonesian way, with dots between thousands.
     *
     * @param amount an amount in whole rupiah
     * @return for example {@code "Rp 28.000"}
     */
    static String rupiah(long amount) {
        return "Rp " + String.format(Locale.ROOT, "%,d", amount).replace(',', '.');
    }
}
```

It promises two methods, `priceOf` and `name`, and its documentation is part of the contract: the price is in **whole rupiah** (a `long`, so that prices compare exactly), it is **never negative**, and a rule may **refuse** an order by throwing `IllegalArgumentException`. Every implementation must honor those sentences, and every caller may rely on them.

The interface also carries a `static` method, `rupiah`, called as `PriceRule.rupiah(...)`, which formats an amount the Indonesian way: `Rp 28.000`. `String.format(Locale.ROOT, "%,d", amount)` inserts commas whatever the computer's language settings, and `replace(',', '.')` turns them into dots.

In Python (**Listing 6.35**) `PriceRule` is an abstract base class, because Profitina Laundry owns every rule and wants an incomplete rule to fail loudly the moment it is created. The formatting helper is a plain module-level function.

Listing 6.35 `price_rule.py`: `PriceRule` as an abstract base class, and the `rupiah` helper.

```
"""The PriceRule abstract base class, and a helper that formats rupiah."""
from abc import ABC, abstractmethod


class PriceRule(ABC):
    """A rule that turns a laundry order into a price.

    Every way Profitina Laundry charges for its work is a subclass of
    PriceRule. Code that only needs a price asks for a PriceRule and
    never learns which rule it received.
    """

    @abstractmethod
    def price_of(self, order):
        """Compute the price of an order.

        Args:
            order: the Order to price.

        Returns:
            The price in whole rupiah (an int), never negative.

        Raises:
            ValueError: if this rule cannot price the order.
        """

    @abstractmethod
    def name(self):
        """Name the rule for the receipt, e.g. "wash & fold"."""


def rupiah(amount):
    """Format an amount the Indonesian way: rupiah(28000) == 'Rp 28.000'."""
    return "Rp " + f"{amount:,}".replace(",", ".")
```

6.16.3 Three basic rules

The per-kilogram rule is the only one with arithmetic worth writing down. If the scale shows w kilograms, the rate is r rupiah per kilogram and the minimum is m kilograms, the price is

$$P_{\text{kilo}} = r \cdot \max(m, \lceil w \rceil) \tag{6.6}$$

where $\lceil w \rceil$, the *ceiling* of w, is the smallest whole number not less than w: $\lceil 3.2 \rceil = 4$, $\lceil 3.0 \rceil = 3$, $\lceil 2.0 \rceil = 2$. Check Equation (6.6) against **Table 6.3**: for $w = 2.0$, $\max(3,2) = 3$ and $P = 7\,000 \times 3 = 21\,000$; for $w = 3.2$, $\max(3,4) = 4$ and $P = 28\,000$. **Listing 6.36** is the formula in Java, line for line; `Math.ceil` returns a `double`, so a cast turns it into a `long`.

Listing 6.36 `PerKiloRule.java`: price by started kilogram, with a minimum.

```
/**
 * Charges by weight: every started kilogram counts, with a minimum.
 */
public class PerKiloRule implements PriceRule {
    private final String name;
    private final long ratePerKg;
    private final int minimumKg;

    /**
     * Creates a per-kilogram rule.
     *
     * @param name      the name printed on the receipt
     * @param ratePerKg the price of one kilogram, in rupiah
     * @param minimumKg the smallest number of kilograms ever charged
     */
    public PerKiloRule(String name, long ratePerKg, int minimumKg) {
        this.name = name;
        this.ratePerKg = ratePerKg;
        this.minimumKg = minimumKg;
    }

    @Override
    public long priceOf(Order order) {
        if (order.getWeightKg() <= 0) {
            throw new IllegalArgumentException(name + " needs a weight");
        }
        long startedKg = (long) Math.ceil(order.getWeightKg());   // 3.2 kg -> 4 kg
        long chargedKg = Math.max(minimumKg, startedKg);
        return ratePerKg * chargedKg;
    }

    @Override
    public String name() {
        return name;
    }
}
```

The other two basic rules are almost trivial, and that is a virtue. **Listing 6.37** multiplies a price per piece by the number of pieces and refuses an order with no pieces; **Listing 6.38** returns the same amount whatever the order.

Listing 6.37 `PerItemRule.java`: price per piece.

```
/**
 * Charges per piece, for garments that are handled one by one.
 */
public class PerItemRule implements PriceRule {
    private final String name;
    private final long pricePerPiece;

    /**
     * Creates a per-piece rule.
     *
     * @param name          the name printed on the receipt
     * @param pricePerPiece the price of one piece, in rupiah
     */
    public PerItemRule(String name, long pricePerPiece) {
        this.name = name;
        this.pricePerPiece = pricePerPiece;
    }

    @Override
    public long priceOf(Order order) {
        if (order.getPieces() <= 0) {
            throw new IllegalArgumentException(name + " needs at least one piece");
        }
        return pricePerPiece * order.getPieces();
    }

    @Override
    public String name() {
        return name;
    }
}
```

Listing 6.38 `FlatRule.java`: one fixed price.

```
/**
 * Charges one fixed amount, whatever the order contains.
 */
public class FlatRule implements PriceRule {
    private final String name;
    private final long amount;

    /**
     * Creates a flat-price rule.
     *
     * @param name   the name printed on the receipt
     * @param amount the fixed price, in rupiah
     */
    public FlatRule(String name, long amount) {
        this.name = name;
        this.amount = amount;
    }

    @Override
    public long priceOf(Order order) {
        return amount;
    }

    @Override
    public String name() {
        return name;
    }
}
```

`FlatRule` has a second career, which we shall use in the tests: because it always returns a known number, it is a perfect **stand-in** for “some rule, I don’t care which, that charges 21 001 rupiah”.

Test writers call such a stand-in a *stub*. Being able to plug a stub in wherever a `PriceRule` is expected is a direct gift of the interface.

6.16.4 Rules that wrap other rules

Express service and the member discount are different in kind: they *modify* a price that another rule computed. So each *holds another* `PriceRule`, called `base`, asks it for its price and adjusts the answer; and since each is itself a `PriceRule`, they can wrap each other in any combination. This is the **decorator** pattern [15]: an object that implements an interface by wrapping another object of the same interface and adding something.

The express rule adds half of the base price. Prices are whole rupiah, so the half is computed with integer division, which drops any fraction; in mathematical notation, with $\lfloor x \rfloor$ (the *floor* of x) meaning the largest whole number not greater than x,

$$P_{\text{express}} = P + \left\lfloor \frac{P}{2} \right\rfloor \tag{6.7}$$

For $P = 28\,000$ this is $28\,000 + 14\,000 = 42\,000$; for an odd $P = 21\,001$ it is $21\,001 + \lfloor 10\,500.5 \rfloor = 21\,001 + 10\,500 = 31\,501$. The member rule takes away p percent, again dropping any fraction of a rupiah from the discount:

$$P_{\text{member}} = P - \left\lfloor \frac{p \cdot P}{100} \right\rfloor \tag{6.8}$$

For $P = 42\,000$ and $p = 10$ this is $42\,000 - \lfloor 420\,000/100 \rfloor = 42\,000 - 4\,200 = 37\,800$. **Listing 6.39** and **Listing 6.40** implement Equation (6.7) and Equation (6.8). Notice that in Java, multiplying before dividing — `normal * percent / 100` — keeps the arithmetic exact; `normal * (percent / 100)` would compute `10 / 100 = 0` in integers and give no discount at all.

Listing 6.39 `ExpressRule.java`: a decorator that adds 50%, rounding down.

```
/**
 * Adds a 50% express surcharge to whatever another rule charges.
 */
public class ExpressRule implements PriceRule {
    private final PriceRule base;

    /**
     * Wraps another rule.
     *
     * @param base the rule that prices the ordinary service
     */
    public ExpressRule(PriceRule base) {
        this.base = base;
    }

    @Override
    public long priceOf(Order order) {
        long normal = base.priceOf(order);
        return normal + normal / 2;          // integer division: half a rupiah is dropped
    }

    @Override
    public String name() {
        return base.name() + " + express";
    }
}
```

Listing 6.40 `MemberDiscountRule.java`: a decorator that takes a percentage off for members, and refuses an impossible percentage.

```
/**
 * Gives members a percentage off whatever another rule charges.
 */
public class MemberDiscountRule implements PriceRule {
    private final PriceRule base;
    private final int percent;

    /**
     * Wraps another rule.
     *
     * @param base    the rule that prices the order before the discount
     * @param percent the discount for members, from 0 to 100
     * @throws IllegalArgumentException if {@code percent} is outside 0..100
     */
    public MemberDiscountRule(PriceRule base, int percent) {
        if (percent < 0 || percent > 100) {
            throw new IllegalArgumentException("percent must be 0..100: " + percent);
        }
        this.base = base;
        this.percent = percent;
    }

    @Override
    public long priceOf(Order order) {
        long normal = base.priceOf(order);
        if (!order.isMember()) {
            return normal;
        }
        return normal - normal * percent / 100;   // the discount is rounded down
    }

    @Override
    public String name() {
        return base.name() + " + member " + percent + "%";
    }
}
```

A complete price is a *chain* of rules. For Budi's 3.2 kg of wash-and-fold, express, as a member, the chain is `new MemberDiscountRule(new ExpressRule(new PerKiloRule("wash & fold", 7_000, 3)), 10)`, and its price is the three formulas applied from the inside out:

$$7\,000 \cdot \max(3, \lceil 3.2 \rceil) = 28\,000 \xrightarrow{\text{express}} 42\,000 \xrightarrow{\text{member}} 37\,800 \tag{6.9}$$

Equation (6.9) is also a test case, as we shall see. The whole design is drawn in **Figure 6.7**.

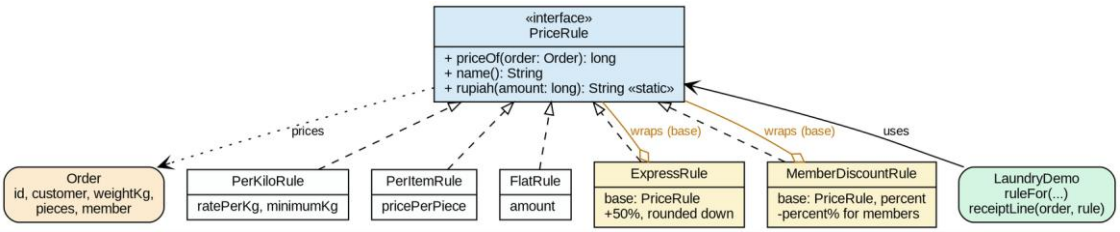


Figure 6.7 The pricing design: five classes implement `PriceRule`, two of them by wrapping another `PriceRule`; the demo program depends only on the interface.

In Python the five rules fit comfortably in one module, **Listing 6.41**. Each subclass of `PriceRule` implements `price_of` and `name`; Python's `//` is floor division, which gives Equation (6.7) and Equation (6.8) directly, and `math.ceil` already returns an `int`.

Listing 6.41 `rules.py`: the five price rules in Python.

```
"""The price rules of Profitina Laundry: five subclasses of PriceRule."""
import math

from price_rule import PriceRule


class PerKiloRule(PriceRule):
    """Charges by weight: every started kilogram counts, with a minimum."""

    def __init__(self, name, rate_per_kg, minimum_kg):
        self._name = name
        self._rate_per_kg = rate_per_kg
        self._minimum_kg = minimum_kg

    def price_of(self, order):
        if order.weight_kg <= 0:
            raise ValueError(f"{self._name} needs a weight")
        started_kg = math.ceil(order.weight_kg)          # 3.2 kg -> 4 kg
        charged_kg = max(self._minimum_kg, started_kg)
        return self._rate_per_kg * charged_kg

    def name(self):
        return self._name


class PerItemRule(PriceRule):
    """Charges per piece, for garments that are handled one by one."""

    def __init__(self, name, price_per_piece):
        self._name = name
        self._price_per_piece = price_per_piece

    def price_of(self, order):
        if order.pieces <= 0:
            raise ValueError(f"{self._name} needs at least one piece")
        return self._price_per_piece * order.pieces

    def name(self):
        return self._name


class FlatRule(PriceRule):
    """Charges one fixed amount, whatever the order contains."""
```

Listing 6.41 (Continued) *rules.py: the five price rules in Python.*

```
    def __init__(self, name, amount):
        self._name = name
        self._amount = amount

    def price_of(self, order):
        return self._amount

    def name(self):
        return self._name


class ExpressRule(PriceRule):
    """Adds a 50% express surcharge to whatever another rule charges."""

    def __init__(self, base):
        self._base = base

    def price_of(self, order):
        normal = self._base.price_of(order)
        return normal + normal // 2          # floor division: half a rupiah is dropped

    def name(self):
        return self._base.name() + " + express"


class MemberDiscountRule(PriceRule):
    """Gives members a percentage off whatever another rule charges."""

    def __init__(self, base, percent):
        if not 0 <= percent <= 100:
            raise ValueError(f"percent must be 0..100: {percent}")
        self._base = base
        self._percent = percent

    def price_of(self, order):
        normal = self._base.price_of(order)
        if not order.member:
            return normal
        return normal - normal * self._percent // 100   # the discount is rounded down

    def name(self):
        return f"{self._base.name()} + member {self._percent}%"
```

Table 6.7 summarizes the rules and their contracts.

Table 6.7 The five price rules of Profitina Laundry.

Rule	Price	Refuses
`PerKiloRule(name, r, m)`	Equation (6.6)	an order with no weight
`PerItemRule(name, price)`	price × pieces	an order with no pieces
`FlatRule(name, amount)`	amount	nothing
`ExpressRule(base)`	Equation (6.7) on the base price	whatever the base refuses
`MemberDiscountRule(base, p)`	Equation (6.8) for members, base price otherwise	p outside 0..100 (when built)

6.16.5 The program that uses the rules

Listing 6.42 prices one morning's orders. Its two methods show both halves of programming to an interface.

Listing 6.42 `LaundryDemo.java`: choosing a rule once, then using it only through the interface.

```
import java.util.List;

/**
 * Prices one morning's orders at Profitina Laundry.
 */
public class LaundryDemo {
    /**
     * Chooses the price rule for a service code.
     *
     * @param service one of WASH, WASH_IRON, DRYCLEAN, SHOES
     * @param express whether the customer wants the order the same day
     * @return the rule, already wrapped for express and member pricing
     * @throws IllegalArgumentException for an unknown service code
     */
    static PriceRule ruleFor(String service, boolean express) {
        PriceRule rule;
        switch (service) {
            case "WASH":
                rule = new PerKiloRule("wash & fold", 7_000, 3);
                break;
            case "WASH_IRON":
                rule = new PerKiloRule("wash & iron", 10_000, 3);
                break;
            case "DRYCLEAN":
                rule = new PerItemRule("dry clean", 25_000);
                break;
            case "SHOES":
                rule = new FlatRule("shoe cleaning", 35_000);
                break;
            default:
                throw new IllegalArgumentException("unknown service: " + service);
        }
        if (express) {
            rule = new ExpressRule(rule);
        }
        return new MemberDiscountRule(rule, 10);   // members always get 10% off
    }
```

Listing 6.42 (Continued) *LaundryDemo.java: choosing a rule once, then using it only through the interface.*

```
    /**
     * Formats one line of the receipt; it works for any PriceRule at all.
     *
     * @param order the order
     * @param rule  the rule that prices it
     * @return the receipt line
     */
    static String receiptLine(Order order, PriceRule rule) {
        return String.format("%-5s %-6s %-36s %10s",
                order.getId(), order.getCustomer(), rule.name(),
                PriceRule.rupiah(rule.priceOf(order)));
    }

    public static void main(String[] args) {
        List<Order> orders = List.of(
                new Order("A001", "Sari", 2.0, 0, false),
                new Order("A002", "Budi", 3.2, 0, true),
                new Order("A003", "Dewi", 0, 4, true),
                new Order("A004", "Eko", 0, 1, false),
                new Order("A005", "Sari", 5.5, 0, false),
                new Order("A006", "Fajar", 2.5, 0, false));
        String[] services = {"WASH", "WASH_IRON", "DRYCLEAN", "SHOES", "WASH", "DRYCLEAN"};
        boolean[] express = {false, true, false, false, true, false};

        long total = 0;
        for (int i = 0; i < orders.size(); i++) {
            Order order = orders.get(i);
            PriceRule rule = ruleFor(services[i], express[i]);
            try {
                System.out.println(receiptLine(order, rule));
                total += rule.priceOf(order);
            } catch (IllegalArgumentException e) {
                System.out.println(order.getId() + "  cannot be priced: " +
e.getMessage());
            }
        }
        System.out.println("Total for the morning: " + PriceRule.rupiah(total));
    }
}
```

Compile all the laundry classes together (`javac Order.java PriceRule.java PerKiloRule.java PerItemRule.java FlatRule.java ExpressRule.java MemberDiscountRule.java LaundryDemo.java`) and run `java LaundryDemo`:

```
A001  Sari   wash & fold + member 10%              Rp 21.000
A002  Budi   wash & iron + express + member 10%    Rp 54.000
A003  Dewi   dry clean + member 10%                Rp 90.000
A004  Eko    shoe cleaning + member 10%            Rp 35.000
A005  Sari   wash & fold + express + member 10%    Rp 63.000
A006  cannot be priced: dry clean needs at least one piece
Total for the morning: Rp 263.000
```

`ruleFor` is the only place in the program that knows the concrete rule classes. It is a small *factory*: it turns a service code into a ready-made chain of rules and returns it with the declared type `PriceRule`. `receiptLine` knows nothing but the interface. It works for every rule that exists today and for every rule anybody writes tomorrow, and it will never need to change for a new price. The `main` loop handles refusals with an ordinary `try`/`catch`: order A006 asked for dry cleaning with a weight but no pieces, `PerItemRule` refused, and the refusal passed unchanged through the member-discount wrapper, exactly as **Table 6.7** promised.

Check the receipt against the formulas. A001: 2.0 kg charged as 3 kg, $3 \times 7\,000 = 21\,000$, not a member. A002: Equation (6.9) again, but at 10 000 per kilogram: $40\,000 \to 60\,000 \to 54\,000$. A003: $4 \times 25\,000 = 100\,000$, minus 10% for a member, 90 000. A004: flat 35 000. A005: 5.5 kg is 6 kg, 42 000, express 63 000. Total $21\,000 + 54\,000 + 90\,000 + 35\,000 + 63\,000 = 263\,000$. The Python program, **Listing 6.43**, prints exactly the same receipt.

Listing 6.43 `laundry_demo.py`: the same morning in Python.

```
"""Prices one morning's orders at Profitina Laundry."""
from order import Order
from price_rule import PriceRule, rupiah
from rules import ExpressRule, FlatRule, MemberDiscountRule, PerItemRule, PerKiloRule


def rule_for(service, express):
    """Choose the price rule for a service code, wrapped for express and members."""
    if service == "WASH":
        rule = PerKiloRule("wash & fold", 7_000, 3)
    elif service == "WASH_IRON":
        rule = PerKiloRule("wash & iron", 10_000, 3)
    elif service == "DRYCLEAN":
        rule = PerItemRule("dry clean", 25_000)
    elif service == "SHOES":
        rule = FlatRule("shoe cleaning", 35_000)
    else:
        raise ValueError(f"unknown service: {service}")
    if express:
        rule = ExpressRule(rule)
    return MemberDiscountRule(rule, 10)     # members always get 10% off


def receipt_line(order, rule: PriceRule):
    """Format one line of the receipt; it works for any PriceRule at all."""
    return (f"{order.id:<5} {order.customer:<6} {rule.name():<36} "
            f"{rupiah(rule.price_of(order)):>10}")


orders = [Order("A001", "Sari", 2.0, 0, False),
          Order("A002", "Budi", 3.2, 0, True),
          Order("A003", "Dewi", 0, 4, True),
          Order("A004", "Eko", 0, 1, False),
          Order("A005", "Sari", 5.5, 0, False),
          Order("A006", "Fajar", 2.5, 0, False)]
services = ["WASH", "WASH_IRON", "DRYCLEAN", "SHOES", "WASH", "DRYCLEAN"]
express = [False, True, False, False, True, False]
```

Listing 6.43 (Continued) *laundry_demo.py: the same morning in Python.*

```
total = 0
for order, service, fast in zip(orders, services, express):
    rule = rule_for(service, fast)
    try:
        print(receipt_line(order, rule))
        total += rule.price_of(order)
    except ValueError as e:
        print(f"{order.id}  cannot be priced: {e}")
print("Total for the morning:", rupiah(total))
```

```
A001  Sari   wash & fold + member 10%              Rp 21.000
A002  Budi   wash & iron + express + member 10%    Rp 54.000
A003  Dewi   dry clean + member 10%                Rp 90.000
A004  Eko    shoe cleaning + member 10%            Rp 35.000
A005  Sari   wash & fold + express + member 10%    Rp 63.000
A006  cannot be priced: dry clean needs at least one piece
Total for the morning: Rp 263.000
```

6.16.6 Testing every rule

A pricing engine that is wrong by one rupiah on one order in a thousand is wrong, and nobody notices for months. So every rule is tested with the techniques of Section 6.8: each equivalence class of **Table 6.3**, both sides of every boundary, every promised refusal, the rounding of Equation (6.7) with an *odd* price, members and non-members, the stack of Equation (6.9), and the formatting helper. **Listing 6.44** is the JUnit 6 test class.

Listing 6.44 `PriceRuleTest.java`: JUnit 6 tests for every price rule.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

/**
 * JUnit 5 tests for every PriceRule of Profitina Laundry.
 */
class PriceRuleTest {
    private final PriceRule wash = new PerKiloRule("wash & fold", 7_000, 3);

    private static Order byWeight(double kg, boolean member) {
        return new Order("T1", "Test", kg, 0, member);
    }

    private static Order byPieces(int pieces, boolean member) {
        return new Order("T2", "Test", 0, pieces, member);
    }

    @Test
    @DisplayName("a light load is charged the 3 kg minimum")
    void perKiloChargesMinimum() {
        assertEquals(21_000, wash.priceOf(byWeight(2.0, false)));
    }

    @Test
    @DisplayName("exactly 3 kg is charged as 3 kg")
    void perKiloExactBoundary() {
        assertEquals(21_000, wash.priceOf(byWeight(3.0, false)));
    }

    @Test
    @DisplayName("a started kilogram counts as a whole kilogram")
    void perKiloRoundsUp() {
        assertEquals(28_000, wash.priceOf(byWeight(3.01, false)));
        assertEquals(28_000, wash.priceOf(byWeight(3.2, false)));
    }

    @Test
    @DisplayName("per-kilo pricing refuses an order with no weight")
    void perKiloRejectsZeroWeight() {
        assertThrows(IllegalArgumentException.class, () -> wash.priceOf(byPieces(2,
false)));
    }
```

Listing 6.44 (Continued) *PriceRuleTest.java: JUnit 6 tests for every price rule.*

```
    @Test
    @DisplayName("per-item pricing multiplies by the number of pieces")
    void perItemMultiplies() {
        PriceRule dryClean = new PerItemRule("dry clean", 25_000);
        assertEquals(100_000, dryClean.priceOf(byPieces(4, false)));
    }

    @Test
    @DisplayName("express adds half, dropping half a rupiah")
    void expressAddsHalfRoundingDown() {
        PriceRule odd = new FlatRule("odd", 21_001);       // a stand-in with an odd price
        assertEquals(31_501, new ExpressRule(odd).priceOf(byPieces(1, false)));
    }

    @Test
    @DisplayName("the member discount applies to members only")
    void memberDiscountOnlyForMembers() {
        PriceRule rule = new MemberDiscountRule(new FlatRule("flat", 50_000), 10);
        assertEquals(45_000, rule.priceOf(byPieces(1, true)));
        assertEquals(50_000, rule.priceOf(byPieces(1, false)));
    }

    @Test
    @DisplayName("a discount outside 0..100 percent is refused")
    void memberDiscountRejectsBadPercent() {
        assertThrows(IllegalArgumentException.class,
                () -> new MemberDiscountRule(wash, 101));
    }

    @Test
    @DisplayName("rules stack: per-kilo, then express, then member")
    void rulesStack() {
        PriceRule rule = new MemberDiscountRule(new ExpressRule(wash), 10);
        assertEquals(37_800, rule.priceOf(byWeight(3.2, true)));
        assertEquals("wash & fold + express + member 10%", rule.name());
    }
```

Listing 6.44 (Continued) *PriceRuleTest.java: JUnit 6 tests for every price rule.*

```
    @Test
    @DisplayName("rupiah amounts get dots between thousands")
    void rupiahFormatting() {
        assertEquals("Rp 0", PriceRule.rupiah(0));
        assertEquals("Rp 999", PriceRule.rupiah(999));
        assertEquals("Rp 1.000", PriceRule.rupiah(1_000));
        assertEquals("Rp 1.234.567", PriceRule.rupiah(1_234_567));
    }
}
```

Two helpers, `byWeight` and `byPieces`, keep the *arrange* step to one short call. Note `expressAddsHalfRoundingDown`: it uses the stub `new FlatRule("odd", 21_001)` instead of a real service rule, so the express rule is tested *alone*. Through the interface it cannot tell the difference — the testing benefit of interfaces promised in Section 6.5.

Run it in VS Code's Testing view, or in a terminal with the jar in `lib` as in Section 6.7.2:

```
javac -cp "lib/*" -d out *.java
java -jar lib/junit-platform-console-standalone-6.1.3.jar execute -cp out --select-class
PriceRuleTest
```

For readers without the JUnit jar, and to show real verdicts here, **Listing 6.45** runs the same checks as a plain program in the style of `DogCheck`, with one extra helper, `checkThrows`, that expects an exception. The code under test is handed to it as a lambda, as JUnit's `assertThrows` is.

Listing 6.45 `PriceRuleCheck.java`: the price-rule tests as a plain program.

```
/**
 * The PriceRuleTest checks as a plain program: no library needed.
 */
public class PriceRuleCheck {
    private static int passed = 0;
    private static int failed = 0;

    static void check(String what, Object expected, Object actual) {
        if (expected.equals(actual)) {
            passed++;
            System.out.println("PASS  " + what);
        } else {
            failed++;
            System.out.println("FAIL  " + what + ": expected " + expected + " but was " +
actual);
        }
    }

    static void checkThrows(String what, Runnable code) {
        try {
            code.run();
            failed++;
            System.out.println("FAIL  " + what + ": no exception was thrown");
        } catch (IllegalArgumentException e) {
            passed++;
            System.out.println("PASS  " + what + " (" + e.getMessage() + ")");
        }
    }

    static Order byWeight(double kg, boolean member) {
        return new Order("T1", "Test", kg, 0, member);
    }

    static Order byPieces(int pieces, boolean member) {
        return new Order("T2", "Test", 0, pieces, member);
    }

    public static void main(String[] args) {
        PriceRule wash = new PerKiloRule("wash & fold", 7_000, 3);
        check("light load pays the 3 kg minimum", 21_000L, wash.priceOf(byWeight(2.0,
false)));
        check("exactly 3 kg is 3 kg", 21_000L, wash.priceOf(byWeight(3.0, false)));
        check("3.01 kg is charged as 4 kg", 28_000L, wash.priceOf(byWeight(3.01, false)));
        check("3.2 kg is charged as 4 kg", 28_000L, wash.priceOf(byWeight(3.2, false)));
        checkThrows("per-kilo refuses zero weight", () -> wash.priceOf(byPieces(2,
false)));
```

Listing 6.45 (Continued) *PriceRuleCheck.java: the price-rule tests as a plain program.*

```
        PriceRule dryClean = new PerItemRule("dry clean", 25_000);
        check("4 pieces of dry cleaning", 100_000L, dryClean.priceOf(byPieces(4, false)));

        PriceRule odd = new FlatRule("odd", 21_001);
        check("express drops half a rupiah", 31_501L, new
ExpressRule(odd).priceOf(byPieces(1, false)));

        PriceRule member = new MemberDiscountRule(new FlatRule("flat", 50_000), 10);
        check("member gets 10% off", 45_000L, member.priceOf(byPieces(1, true)));
        check("non-member pays full price", 50_000L, member.priceOf(byPieces(1, false)));
        checkThrows("101% discount is refused", () -> new MemberDiscountRule(wash, 101));

        PriceRule stacked = new MemberDiscountRule(new ExpressRule(wash), 10);
        check("stacked rules", 37_800L, stacked.priceOf(byWeight(3.2, true)));
        check("stacked name", "wash & fold + express + member 10%", stacked.name());

        check("Rp 0", "Rp 0", PriceRule.rupiah(0));
        check("Rp 1.234.567", "Rp 1.234.567", PriceRule.rupiah(1_234_567));

        System.out.println(passed + " passed, " + failed + " failed");
        if (failed > 0) {
            System.exit(1);
        }
    }
}
```

```
PASS  light load pays the 3 kg minimum
PASS  exactly 3 kg is 3 kg
PASS  3.01 kg is charged as 4 kg
PASS  3.2 kg is charged as 4 kg
PASS  per-kilo refuses zero weight (wash & fold needs a weight)
PASS  4 pieces of dry cleaning
PASS  express drops half a rupiah
PASS  member gets 10% off
PASS  non-member pays full price
PASS  101% discount is refused (percent must be 0..100: 101)
PASS  stacked rules
PASS  stacked name
PASS  Rp 0
PASS  Rp 1.234.567
14 passed, 0 failed
```

> ⚠ **TRAP.** The expected values in **Listing 6.45** are written `21_000L`, with an `L`, and not by accident. `check` takes two `Object`s, so Java *boxes* both numbers: the actual price, a `long`, becomes a `Long`; a plain `21_000` would become an `Integer`. And `Integer.valueOf(21000).equals(Long.valueOf(21000))` is **false** — `equals` returns false for objects of different classes, as the contract of Section 6.9 requires. Without the `L`, every check would fail with the baffling message "expected 21000 but was 21000". JUnit's `assertEquals(long, long)` avoids the trap by comparing primitive values.

The Python tests use pytest, and use one feature JUnit keeps in a separate module: **parametrized tests**. The decorator `@pytest.mark.parametrize` runs one test function once per row of a table, so the four rows of **Table 6.3** become four tests with one body. **Listing 6.46** is the file.

Listing 6.46 `test_price_rules.py`: pytest tests for every price rule, with parametrized boundaries.

```
"""pytest tests for every PriceRule of Profitina Laundry."""
import pytest

from order import Order
from price_rule import rupiah
from rules import ExpressRule, FlatRule, MemberDiscountRule, PerItemRule, PerKiloRule

WASH = PerKiloRule("wash & fold", 7_000, 3)


def by_weight(kg, member=False):
    return Order("T1", "Test", kg, 0, member)


def by_pieces(pieces, member=False):
    return Order("T2", "Test", 0, pieces, member)


@pytest.mark.parametrize("kg, expected", [
    (2.0, 21_000),      # below the minimum
    (3.0, 21_000),      # exactly on the boundary
    (3.01, 28_000),     # just past it: a started kilogram counts
    (3.2, 28_000),
])
def test_per_kilo_minimum_and_rounding(kg, expected):
    assert WASH.price_of(by_weight(kg)) == expected


def test_per_kilo_rejects_zero_weight():
    with pytest.raises(ValueError, match="needs a weight"):
        WASH.price_of(by_pieces(2))


def test_per_item_multiplies():
    assert PerItemRule("dry clean", 25_000).price_of(by_pieces(4)) == 100_000


def test_express_adds_half_rounding_down():
    odd = FlatRule("odd", 21_001)            # a stand-in with an odd price
    assert ExpressRule(odd).price_of(by_pieces(1)) == 31_501
```

Listing 6.46 (Continued) *test_price_rules.py: pytest tests for every price rule, with parametrized boundaries.*

```
def test_member_discount_only_for_members():
    rule = MemberDiscountRule(FlatRule("flat", 50_000), 10)
    assert rule.price_of(by_pieces(1, member=True)) == 45_000
    assert rule.price_of(by_pieces(1, member=False)) == 50_000


def test_member_discount_rejects_bad_percent():
    with pytest.raises(ValueError):
        MemberDiscountRule(WASH, 101)


def test_rules_stack():
    rule = MemberDiscountRule(ExpressRule(WASH), 10)
    assert rule.price_of(by_weight(3.2, member=True)) == 37_800
    assert rule.name() == "wash & fold + express + member 10%"


@pytest.mark.parametrize("amount, text", [
    (0, "Rp 0"), (999, "Rp 999"), (1_000, "Rp 1.000"), (1_234_567, "Rp 1.234.567"),
])
def test_rupiah_formatting(amount, text):
    assert rupiah(amount) == text
```

`python3 -m pytest -v test_price_rules.py` gives one line per test, each parameter row listed separately (the header, which names your Python version and folder, is left out here):

```
test_price_rules.py::test_per_kilo_minimum_and_rounding[2.0-21000] PASSED [  7%]
test_price_rules.py::test_per_kilo_minimum_and_rounding[3.0-21000] PASSED [ 14%]
test_price_rules.py::test_per_kilo_minimum_and_rounding[3.01-28000] PASSED [ 21%]
test_price_rules.py::test_per_kilo_minimum_and_rounding[3.2-28000] PASSED [ 28%]
test_price_rules.py::test_per_kilo_rejects_zero_weight PASSED            [ 35%]
test_price_rules.py::test_per_item_multiplies PASSED                     [ 42%]
test_price_rules.py::test_express_adds_half_rounding_down PASSED         [ 50%]
test_price_rules.py::test_member_discount_only_for_members PASSED        [ 57%]
test_price_rules.py::test_member_discount_rejects_bad_percent PASSED     [ 64%]
test_price_rules.py::test_rules_stack PASSED                             [ 71%]
test_price_rules.py::test_rupiah_formatting[0-Rp 0] PASSED               [ 78%]
test_price_rules.py::test_rupiah_formatting[999-Rp 999] PASSED           [ 85%]
test_price_rules.py::test_rupiah_formatting[1000-Rp 1.000] PASSED        [ 92%]
test_price_rules.py::test_rupiah_formatting[1234567-Rp 1.234.567] PASSED [100%]
============================== 14 passed in 0.02s ==============================
```

To see what pytest does when a test fails, we changed the last line of `ExpressRule.price_of` to `return round(normal * 1.5)` — a plausible "improvement" that rounds to the nearest rupiah instead of down — and ran pytest again (`-q` for a quiet summary). The relevant part of the report:

```
......F.......                                                          [100%]
=================================== FAILURES ===================================
_____________________ test_express_adds_half_rounding_down _____________________

    def test_express_adds_half_rounding_down():
        odd = FlatRule("odd", 21_001)            # a stand-in with an odd price
>       assert ExpressRule(odd).price_of(by_pieces(1)) == 31_501
E       assert 31502 == 31501
E        +  where 31502 = price_of(Order[T2, Test, 0 kg, 1 pcs])
```

pytest marks the failing line with `>` and takes the expression apart: the left side was `31502`, from `price_of` on `Order[T2, Test, 0 kg, 1 pcs]` — our `__repr__` making the report readable. 21 001 × 1.5 = 31 501.5, and Python's `round` rounds a half to the nearest *even* number, 31 502. Whether the laundry wants that is a business decision; what matters is that the change cannot slip in unnoticed, and the odd price was chosen precisely for that.

Finally, **Listing 6.47** is the plain Python checker, matching **Listing 6.45** line for line.

Listing 6.47 `price_rule_check.py`: the price-rule tests as a plain Python program.

```
"""The test_price_rules checks as a plain program: no library needed."""
import sys

from order import Order
from price_rule import rupiah
from rules import ExpressRule, FlatRule, MemberDiscountRule, PerItemRule, PerKiloRule

passed = 0
failed = 0


def check(what, expected, actual):
    global passed, failed
    if expected == actual:
        passed += 1
        print("PASS ", what)
    else:
        failed += 1
        print("FAIL ", what + f": expected {expected!r} but was {actual!r}")


def check_raises(what, code):
    global passed, failed
    try:
        code()
        failed += 1
        print("FAIL ", what + ": no exception was raised")
    except ValueError as e:
        passed += 1
        print("PASS ", what, f"({e})")


def by_weight(kg, member=False):
    return Order("T1", "Test", kg, 0, member)


def by_pieces(pieces, member=False):
    return Order("T2", "Test", 0, pieces, member)
```

Listing 6.47 (Continued) *price_rule_check.py: the price-rule tests as a plain Python program.*

```
wash = PerKiloRule("wash & fold", 7_000, 3)
check("light load pays the 3 kg minimum", 21_000, wash.price_of(by_weight(2.0)))
check("exactly 3 kg is 3 kg", 21_000, wash.price_of(by_weight(3.0)))
check("3.01 kg is charged as 4 kg", 28_000, wash.price_of(by_weight(3.01)))
check("3.2 kg is charged as 4 kg", 28_000, wash.price_of(by_weight(3.2)))
check_raises("per-kilo refuses zero weight", lambda: wash.price_of(by_pieces(2)))

dry_clean = PerItemRule("dry clean", 25_000)
check("4 pieces of dry cleaning", 100_000, dry_clean.price_of(by_pieces(4)))

odd = FlatRule("odd", 21_001)
check("express drops half a rupiah", 31_501, ExpressRule(odd).price_of(by_pieces(1)))

member = MemberDiscountRule(FlatRule("flat", 50_000), 10)
check("member gets 10% off", 45_000, member.price_of(by_pieces(1, member=True)))
check("non-member pays full price", 50_000, member.price_of(by_pieces(1)))
check_raises("101% discount is refused", lambda: MemberDiscountRule(wash, 101))

stacked = MemberDiscountRule(ExpressRule(wash), 10)
check("stacked rules", 37_800, stacked.price_of(by_weight(3.2, member=True)))
check("stacked name", "wash & fold + express + member 10%", stacked.name())

check("Rp 0", "Rp 0", rupiah(0))
check("Rp 1.234.567", "Rp 1.234.567", rupiah(1_234_567))

print(f"{passed} passed, {failed} failed")
if failed > 0:
    sys.exit(1)
```

```
PASS  light load pays the 3 kg minimum
PASS  exactly 3 kg is 3 kg
PASS  3.01 kg is charged as 4 kg
PASS  3.2 kg is charged as 4 kg
PASS  per-kilo refuses zero weight (wash & fold needs a weight)
PASS  4 pieces of dry cleaning
PASS  express drops half a rupiah
PASS  member gets 10% off
PASS  non-member pays full price
PASS  101% discount is refused (percent must be 0..100: 101)
PASS  stacked rules
PASS  stacked name
PASS  Rp 0
PASS  Rp 1.234.567
14 passed, 0 failed
```

6.16.7 What the interface bought us

Count what it would take to add the rainy-season promotion — say, 5% off every order in January. One new class, `SeasonalRule`, that wraps a base rule, exactly like `MemberDiscountRule`; one line in `ruleFor`; and a handful of tests. `Order`, `PriceRule`, the five existing rules, `receiptLine`, and every existing test stay untouched — and the existing tests go on proving that the old prices did not change. Exercise 6.11 asks you to add a rule of this kind.

That is the whole point of the chapter in one paragraph: the *documentation* of `PriceRule` says what every rule promises, the *tests* check that each rule keeps the promise, and the *interface* lets the rest of the program rely on the promise without knowing which rule it holds.

6.17 Profitina in Practice

Profitina, the AI-visibility platform that runs at app.profitina.com, measures how ready a website is to be quoted by AI answer engines, and it does so from many independent signals: whether the page's content can be extracted cleanly, whether it carries structured data, how technically sound and trusted the site is, and more. The shape of that part of the product is the shape of Section 6.16. Every signal is a separate class behind **one interface** — roughly "given a scanned site, produce a score and an explanation" — and the code that assembles a report asks only for that interface. A new signal is a new class; the report code, the dashboard and the weekly trend charts do not change. (How each signal is scored is the product's own business; the *structure* is the lesson.)

The same idea protects the product where it meets things it does not control. Profitina runs its language model locally (Qwen2.5 through Ollama), and code that needs a model's answer is best written against a narrow interface: "given a prompt, return an answer, or say that none is available". In tests, a stand-in that returns a fixed answer takes the model's place, exactly as `FlatRule` stood in for a real price rule, so the logic around the model is tested in milliseconds. In production, when no model is available, Profitina says so plainly instead of passing off a template as an AI answer: a stand-in is for tests, never for fooling users.

Testing is a release gate there. The full automated suite — well over a thousand tests, in Python for the server and in JavaScript for the user interface — must be green before a release goes out, and a release counts as done only when the newly deployed version answers its health check. It is `DogCheck`'s non-zero exit status applied to a whole product: a machine, not a mood, decides whether the code is ready.

Documentation and equality appear too. The server is written in Python with FastAPI, which builds interactive API reference pages directly from type annotations and docstrings — `pydoc`'s idea, grown up. When a site is crawled, addresses such as `https://Example.com/` and `https://example.com` must count as the same page, which is `Customer`'s normalized phone number in another costume; get `equals` and `hashCode` wrong there and the same page is scanned and scored twice. And because Profitina follows Indonesia's personal-data protection law, UU PDP 27/2022, what a class shows about itself in its printed form is a deliberate design decision, as Section 6.10 warned.

6.18 Common Mistakes, and How to Avoid Them

Ending a Javadoc comment with `/` instead of `*/`. The comment swallows everything up to the next `*/`, possibly a whole method, and the compiler says nothing (Section 6.3.1). *Avoid it:* use an editor with syntax coloring, and generate the documentation regularly.

`@param` names that do not match the parameters. After a rename, the comment still describes the old name. *Avoid it:* run `javadoc` and treat its errors and warnings as bugs; in Python, rename in the docstring at the same moment.

Documentation that describes the implementation. "Loops over the array and uses `Math.pow`" helps no caller and becomes false the day the loop is rewritten. *Avoid it:* describe purpose, parameters, result, failures and units.

A test that recomputes the answer with the same formula. It agrees with every bug in the formula. *Avoid it:* write expected values as literals worked out by hand.

Tests that depend on each other or on their order. *Avoid it:* build fresh objects in `@BeforeEach`/`setUp`, or inside each test.

Only testing the pleasant cases. Most bugs live at boundaries and in error handling. *Avoid it:* list the equivalence classes, test both sides of every boundary, and test every exception the documentation promises.

Comparing doubles with `==`, or swapping expected and actual. `0.1 + 0.2 == 0.3` is false, and `assertEquals(rover.getAge(), 6)` lies when it fails. *Avoid it:* compare with a tolerance (Equation (6.2)) or keep money in whole units; put the expected value first.

`equals` without `hashCode`, or `equals(Customer other)`. The first makes hash sets and maps treat equal objects as different; the second overloads instead of overriding. *Avoid it:* override both methods from the same fields, take an `Object` parameter, write `@Override`, and compile with `-Xlint:all`.

Using `==` to compare strings or other objects in Java. It tests identity, not content. *Avoid it:* `a.equals(b)` for objects; `==` only for primitives.

Leaving the default `toString`, or putting secrets in it. `PlainCustomer@1b6d3586` tells you nothing, and a password in a log is a leak. *Avoid it:* override `toString`/`__repr__` deliberately in every class you might print.

Overloading one name for arguments that mean different things, or trying to overload in Python. `describe(3L)` becomes kilograms and `describe('A')` 65 pieces; a second Python `def` silently replaces the first. *Avoid it:* give different meanings different names; in Python use default arguments, `*args` or `singledispatch`.

Assuming `compareTo` returns −1, 0 or 1. Only the sign is promised (Equation (6.5)). *Avoid it:* test `< 0`, `== 0`, `> 0`, and implement the generic `Comparable<Name>`.

Declaring variables with the class type when an interface would do. `ArrayList<Order> orders` or `TreeDB db` ties the code to one implementation and brings back the flag-testing of Section 6.13.1. *Avoid it:* declare `List<Order>`, `DBops`, `PriceRule`; name the concrete class only where the object is created.

Rounding money in the wrong place. `normal * (percent / 100)` gives zero in integers, and `round` instead of floor changes prices by a rupiah. *Avoid it:* write the rounding rule into the documentation and test it with an odd amount.

6.19 Chapter Summary

- A class lists its fields, constructors and methods. Java allows several constructors (overloads); Python has one `__init__` and uses default arguments or `@classmethod` factories.
- **Documentation comments** describe what a class or method promises. Javadoc comments start with `/**` and use tags such as `@param`, `@return` and `@throws` (**Table 6.1**); Python docstrings follow PEP 257, here in Google style. `javadoc` builds and checks HTML pages, `pydoc` shows or writes a module's documentation, and `doctest` runs the examples in docstrings (**Figure 6.1**). A fixed **seed** makes random numbers repeatable, which tests need.
- A **unit test** runs one unit on a chosen input and compares the actual result with an expected result worked out beforehand, in three steps: **arrange, act, assert** (**Figure 6.2**). A plain program with a `check` method and a non-zero exit status is already a test suite.

- **JUnit 6** (`@Test`, `@BeforeEach`, `@DisplayName`, `assertEquals`, `assertThrows`), **unittest** and **pytest** do the same job (**Table 6.2**); doubles are compared with a tolerance (Equation (6.2)).
- Good tests are chosen by **equivalence classes and boundaries** (**Table 6.3**) and are independent of each other. Writing the test first — **red, green, refactor** (**Figure 6.3**) — makes you design from the caller's side.
- **Identity** is not **equality** (**Figure 6.4**). Override `equals` and `hashCode` together, from the same fields, so that equal objects have equal hash codes (Equation (6.3)). `toString` gives an object its printed form; Python separates `__repr__` (for developers) from `__str__` (for users).
- **Overloading** gives one name several parameter lists, chosen at compile time by exact match, then widening, then boxing (**Listing 6.18**). Python cannot overload; it uses default arguments, `*args` and `singledispatch` (**Table 6.4**).
- An **interface** lists methods without bodies; a class `implements` it, the interface serves as a type, and calls are dispatched to the actual object's class at run time. Coding a client against an interface removes flag-testing (**Figure 6.5**); **default methods** share code; up-casting is automatic and down-casting needs a checked cast (**Figure 6.6**).
- The Java library is built on interfaces such as `Comparable`, `Iterable`, `List` and `ActionListener` (**Table 6.5**); `compareTo` promises only a sign (Equation (6.5)). Python states interfaces as **abstract base classes** or **protocols** (**Table 6.6**).
- **Program to an interface, not an implementation.** Profitina Laundry's prices are five classes behind `PriceRule` (**Figure 6.7**, **Table 6.7**), two of them decorators; receipts know only the interface, every rule is tested alone, and a new rule is a new class and nothing else.

6.20 Exercises

The exercises follow the order of the chapter. Those with a program have a complete Java and Python solution in Section 6.21; the others are answered in words. Try each one before you look.

Exercise 6.1 (documenting a method). Profitina Laundry needs a small helper, `kilosCharged(weightKg, minimumKg)`, that returns the number of kilograms an order is charged for: every started kilogram counts, and never fewer than the minimum; a weight of zero or less is refused. Write the method in Java with a complete Javadoc comment (summary, two examples, every `@param`, `@return`, `@throws`) and in Python with a Google-style docstring whose examples are doctests, including the refusal. Run `javadoc` and `python3 -m doctest -v` and make both run clean.

Exercise 6.2 (true or false?). Decide, with a reason, whether each statement is true. (a) A comment that starts with `/*` is a Javadoc comment. (b) The Java compiler keeps Javadoc comments in the `.class` file so that `javadoc` can read them later. (c) A Python docstring can be read while the program runs. (d) The documentation of a method should explain which loop it uses, so that callers know how fast it is. (e) If `javadoc` prints no warnings, the documentation is correct.

Exercise 6.3 (one constructor does the work). Rewrite `BankAccount` from **Listing 6.1** as `ChainedAccount`, so that both public constructors delegate with `this(...)` to one private constructor that sets the fields, and give it a `toString`. In Python, keep a single `__init__(balance, overdraft)` and offer the two ways of opening an account as class methods `empty()` and `with_deposit(amount)`. Show that an account opened with a deposit of 0 gets the overdraft of 50.

Exercise 6.4 (testing doubles). Write unit tests for `interestOn` / `interest_on` from Section 6.2: zero years, one year (which must equal simple interest), the two-year example of Equation (6.1), a zero rate, and the refusal of negative years. Use JUnit 6 in Java and `unittest` in Python, and compare with a tolerance. For each test, say which equivalence class or boundary it represents.

Exercise 6.5 (what is wrong with these tests?). Each test below passes today. Explain why each one is nevertheless a poor test, and rewrite it.

```
(a) assertEquals(7000 * Math.max(3, (long) Math.ceil(3.2)), wash.priceOf(order));
(b) static Dog shared = new Dog("Rover", 5);
    @Test void birthday() { shared.incrementAge(); assertEquals(6, shared.getAge()); }
    @Test void age()      { assertEquals(5, shared.getAge()); }
(c) assertEquals(rover.getAge(), 6);
(d) def test_prices(): assert rules_work()
```

Exercise 6.6 (test first: a loyalty card). Members of Profitina Laundry earn one point for every full 10 000 rupiah they pay; 9 999 rupiah earns nothing, and a negative amount is an error. Following the red–green–refactor cycle of **Figure 6.3**, write the tests *first* — a plain checker `MemberCardCheck` in Java and a pytest file `test_member_card.py` — choosing the boundaries deliberately; run them and watch them fail; then write `MemberCard.pointsFor` and `member_card.points_for`.

Exercise 6.7 (equality with two fields). The laundry's washing machines are identified by number *and* brand (machine 3 from two different suppliers are two machines). Write a class `Machine` with `equals`/`hashCode` (using `Objects.hash`) and `__eq__`/`__hash__`, and show that when two staff members report the same broken machine, a set of machines to repair holds it once.

Exercise 6.8 (records and data classes). Write `CustomerRecord` as a Java `record` and as a Python `@dataclass(frozen=True)` with the fields `name` and `phone`. Put "Sari" and "Sari W." with the same phone into a set. How many elements does the set hold, and why does `Customer` from **Listing 6.15** give a different answer?

Exercise 6.9 (overloading puzzle). Predict, *before* running it, what each line prints. Java: three overloads `describe(int)`, `describe(double)`, `describe(Object)`, called with a `short`, `3f`, `Integer.valueOf(3)`, `"3"` and `'3'`. Python: a `singledispatch` function `describe` with versions for `int` and `float`, called with `3`, `3.0`, `False`, `'3'`, `Fraction(3, 1)` and `2 ** 100`. Then run the programs and explain every surprise.

Exercise 6.10 (making a class fit an interface). Use the `average` method of **Listing 6.22** and **Listing 6.23** to compute the average weight of three orders, *without changing* `Order` and without changing `average`.

Exercise 6.11 (a new price rule). Profitina Laundry introduces a bulk discount: an order *strictly heavier* than a threshold gets a percentage off the price computed by another rule. Write `BulkDiscountRule` as a decorator in the style of `MemberDiscountRule`, refuse a percentage outside 0..100, and test it with the boundaries 9 kg, exactly 10 kg, 10.01 kg and 12 kg for a threshold of 10 kg, 20% off, on wash-and-fold. Work the expected prices out by hand first.

Exercise 6.12 (does the order of decorators matter?). A member ordering express service could be priced as `MemberDiscountRule(ExpressRule(base))` or as `ExpressRule(MemberDiscountRule(base))`. In exact arithmetic both give $1.5 \times 0.9 = 1.35$ times the base price. Write a program that tries every base price from 21 000 to 21 020 rupiah with a `FlatRule` and prints the bases where the two orders disagree. Explain the result, and say what the laundry should do about it.

Exercise 6.13 (a Comparator). Sort a list of orders heaviest first, and orders of equal weight by ticket id, using an implementation of the interface `java.util.Comparator<Order>` in Java and a `key=` function in Python. Why is `Comparator` a better choice here than making `Order` implement `Comparable`?

6.21 Solutions to the Exercises

Solution 6.1. The helper is Equation (6.6) without the rate. The comment states the contract — started kilograms, the minimum, the refusal — and nothing about `Math.ceil`, which is an implementation detail. **Listing 6.48** is the Java version.

Listing 6.48 `KilosCharged.java`: a fully documented helper, with the documented examples in `main`.

```
/**
 * Works out how many kilograms a per-kilo laundry order is charged for.
 */
public class KilosCharged {
    private KilosCharged() {
        // only static methods: no objects of this class are needed
    }

    /**
     * Returns the number of kilograms to charge: every started kilogram
     * counts, and never fewer than the minimum.
     *
     * <p>For example, {@code kilosCharged(3.2, 3)} is 4 and
     * {@code kilosCharged(1.5, 3)} is 3.</p>
     *
     * @param weightKg  the weight on the scale, in kilograms; must be positive
     * @param minimumKg the smallest number of kilograms ever charged
     * @return the number of kilograms to charge
     * @throws IllegalArgumentException if {@code weightKg} is zero or negative
     */
    public static long kilosCharged(double weightKg, int minimumKg) {
        if (weightKg <= 0) {
            throw new IllegalArgumentException("weight must be positive: " + weightKg);
        }
        return Math.max(minimumKg, (long) Math.ceil(weightKg));
    }

    /**
     * Prints the two documented examples.
     *
     * @param args not used
     */
    public static void main(String[] args) {
        System.out.println(kilosCharged(3.2, 3));
        System.out.println(kilosCharged(1.5, 3));
    }
}
```

```
4
3
```

`javadoc -d docs KilosCharged.java` runs without a warning. The private constructor matters here: the class has only static methods, and without it the compiler would add a public default constructor that `javadoc` reports as undocumented. The Python version, **Listing 6.49**, turns the examples into doctests; the third shows how a doctest expects an exception: the `Traceback` line, a line of dots for the parts that vary, and the last line of the error message.

Listing 6.49 `kilos_charged.py`: a Google-style docstring whose examples are doctests, one of them an expected exception.

```
"""How many kilograms a per-kilo laundry order is charged for."""
import math


def kilos_charged(weight_kg, minimum_kg):
    """Return the number of kilograms to charge.

    Every started kilogram counts, and never fewer than the minimum.

    Args:
        weight_kg: the weight on the scale in kilograms; must be positive.
        minimum_kg: the smallest number of kilograms ever charged.

    Returns:
        The number of kilograms to charge, as an int.

    Raises:
        ValueError: if weight_kg is zero or negative.

    >>> kilos_charged(3.2, 3)
    4
    >>> kilos_charged(1.5, 3)
    3
    >>> kilos_charged(0, 3)
    Traceback (most recent call last):
        ...
    ValueError: weight must be positive: 0
    """
    if weight_kg <= 0:
        raise ValueError(f"weight must be positive: {weight_kg}")
    return max(minimum_kg, math.ceil(weight_kg))


if __name__ == "__main__":
    print(kilos_charged(3.2, 3))
    print(kilos_charged(1.5, 3))
```

```
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```

Solution 6.2. (a) False: a Javadoc comment starts with `/**`; `/*` begins an ordinary block comment. (b) False: the compiler discards all comments, and `javadoc` reads the `.java` files. (c) True: it is stored in the attribute `__doc__`, and `help()` prints it. (d) False: documentation states what the method promises — parameters, result, failures, units, and any speed guarantee callers may rely on — not which loop it uses. (e) False: `javadoc` checks the *form*, not the *truth*; "returns the balance" on a method that returns the overdraft passes silently. Only a reader, or a test, catches that.

Solution 6.3. **Listing 6.50** puts all the field-setting in one private constructor. `this(...)` must be the first statement of a constructor, and the private constructor can take parameters that no outside caller should combine freely.

Listing 6.50 `ChainedAccount.java`: two public constructors that delegate to one private constructor.

```java
/**
 * BankAccount with constructor chaining: both public constructors
 * delegate to one private constructor that does all the work.
 */
public class ChainedAccount {
    private final double balance;
    private final int overdraft;

    /**
     * Opens an account with an opening deposit and an overdraft of 50.
     *
     * @param openingAmount the money paid in on the first day
     */
    public ChainedAccount(double openingAmount) {
        this(openingAmount, 50);          // must be the first statement
    }

    /**
     * Opens an empty account with an overdraft of 10.
     */
    public ChainedAccount() {
        this(0, 10);
    }

    private ChainedAccount(double balance, int overdraft) {
        this.balance = balance;           // the only place the fields are set
        this.overdraft = overdraft;
    }

    @Override
    public String toString() {
        return "ChainedAccount[balance=" + balance + ", overdraft=" + overdraft + "]";
    }

    public static void main(String[] args) {
        System.out.println(new ChainedAccount());
        System.out.println(new ChainedAccount(25));
        System.out.println(new ChainedAccount(0));
    }
}
```

```
ChainedAccount[balance=0.0, overdraft=10]
ChainedAccount[balance=25.0, overdraft=50]
ChainedAccount[balance=0.0, overdraft=50]
```

In Python (**Listing 6.51**), a *class method* receives the class itself as `cls`, and `cls(0, 10)` calls `__init__`. The two class methods are *named* constructors, which say more to a reader than `BankAccount()` did.

Listing 6.51 `chained_account.py`: alternative constructors written as class methods.

```python
"""Alternative constructors in Python: class methods that call the one __init__."""


class ChainedAccount:
    def __init__(self, balance, overdraft):
        self._balance = float(balance)      # the only place the attributes are set
        self._overdraft = overdraft

    @classmethod
    def empty(cls):
        """Open an empty account with an overdraft of 10."""
        return cls(0, 10)

    @classmethod
    def with_deposit(cls, opening_amount):
        """Open an account with a deposit and an overdraft of 50."""
        return cls(opening_amount, 50)

    def __repr__(self):
        return f"ChainedAccount[balance={self._balance}, overdraft={self._overdraft}]"


print(ChainedAccount.empty())
print(ChainedAccount.with_deposit(25))
print(ChainedAccount.with_deposit(0))
```

```
ChainedAccount[balance=0.0, overdraft=10]
ChainedAccount[balance=25.0, overdraft=50]
ChainedAccount[balance=0.0, overdraft=50]
```

The third line of both outputs answers the last question: a deposit of zero is still a deposit, so the overdraft is 50.

Solution 6.4. The equivalence classes are: no time at all (zero years), the simplest real case (one year, where compound and simple interest coincide: $1\,000 \times 0.06 = 60$), the general case (two years, Equation (6.1)), a degenerate rate (zero), and the invalid class (negative years). **Listing 6.52** is the JUnit 6 class; each `assertEquals` has a third argument, the tolerance of Equation (6.2).

Listing 6.52 `BankAccountTest.java`: JUnit 6 tests for `interestOn`, comparing doubles with a delta.

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

import org.junit.jupiter.api.Test;

/**
 * JUnit 5 tests for BankAccount.interestOn, comparing doubles with a delta.
 */
class BankAccountTest {
    private final BankAccount account = new BankAccount();

    @Test
    void zeroYearsEarnNothing() {
        assertEquals(0.0, account.interestOn(1_000, 0, 0.06), 1e-9);
    }

    @Test
    void oneYearIsSimpleInterest() {
        assertEquals(60.0, account.interestOn(1_000, 1, 0.06), 1e-9);
    }

    @Test
    void twoYearsCompound() {
        assertEquals(123_600.0, account.interestOn(1_000_000, 2, 0.06), 1e-6);
    }

    @Test
    void zeroRateEarnsNothing() {
        assertEquals(0.0, account.interestOn(1_000, 5, 0.0), 1e-9);
    }

    @Test
    void negativeYearsAreRejected() {
        assertThrows(IllegalArgumentException.class, () -> account.interestOn(1_000, -1,
0.06));
    }
}
```

The tolerance for the two-year case is 10^{-6} rather than 10^{-9} because the numbers are a million times larger, and so is the rounding noise. **Listing 6.53** is the `unittest` version.

Listing 6.53 `test_bank_account.py`: `unittest` tests for `interest_on` with `assertAlmostEqual`.

```
"""unittest tests for BankAccount.interest_on, comparing floats with a tolerance."""
import unittest

from bank_account import BankAccount


class InterestTest(unittest.TestCase):
    def setUp(self):
        self.account = BankAccount()

    def test_zero_years_earn_nothing(self):
        self.assertAlmostEqual(0.0, self.account.interest_on(1_000, 0, 0.06), delta=1e-9)

    def test_one_year_is_simple_interest(self):
        self.assertAlmostEqual(60.0, self.account.interest_on(1_000, 1, 0.06), delta=1e-9)

    def test_two_years_compound(self):
        self.assertAlmostEqual(123_600.0, self.account.interest_on(1_000_000, 2, 0.06),
                               delta=1e-6)

    def test_zero_rate_earns_nothing(self):
        self.assertAlmostEqual(0.0, self.account.interest_on(1_000, 5, 0.0), delta=1e-9)

    def test_negative_years_are_rejected(self):
        with self.assertRaises(ValueError):
            self.account.interest_on(1_000, -1, 0.06)


if __name__ == "__main__":
    unittest.main()
```

```
test_negative_years_are_rejected
(test_bank_account.InterestTest.test_negative_years_are_rejected) ... ok
test_one_year_is_simple_interest
(test_bank_account.InterestTest.test_one_year_is_simple_interest) ... ok
test_two_years_compound (test_bank_account.InterestTest.test_two_years_compound) ... ok
test_zero_rate_earns_nothing (test_bank_account.InterestTest.test_zero_rate_earns_nothing)
... ok
test_zero_years_earn_nothing (test_bank_account.InterestTest.test_zero_years_earn_nothing)
... ok

----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

Solution 6.5. (a) The expected value uses the code's own formula, so a bug in the formula (`Math.floor` for `Math.ceil`) appears on both sides. Write the literal `28_000`. (b) The tests share one dog, so `age` fails whenever `birthday` happens to run first. Build the dog in `@BeforeEach`. (c) Expected and actual are swapped, so a failure would report "expected: <7> but was: <6>". Write `assertEquals(6, rover.getAge())`. (d) One test checks everything through a helper that returns a boolean, so a failure can only say `assert False`. Write one test per behavior, each asserting a concrete value.

Solution 6.6. The tests come first, and choosing them is the real work: zero, the value just below the first point, exactly the first point, just above it, one typical order, and the refusal. **Listing 6.54** is the Java checker; before `MemberCard` exists it does not even compile, and that is the first "red".

Listing 6.54 `MemberCardCheck.java`: the loyalty-card tests, written before the code they test.

```
/**
 * The tests for MemberCard, written BEFORE MemberCard itself.
 */
public class MemberCardCheck {
    private static int passed = 0;
    private static int failed = 0;

    static void check(String what, long expected, long actual) {
        if (expected == actual) {
            passed++;
            System.out.println("PASS  " + what);
        } else {
            failed++;
            System.out.println("FAIL  " + what + ": expected " + expected + " but was " +
actual);
        }
    }

    public static void main(String[] args) {
        check("nothing paid, no points", 0, MemberCard.pointsFor(0));
        check("just below one point", 0, MemberCard.pointsFor(9_999));
        check("exactly one point", 1, MemberCard.pointsFor(10_000));
        check("just above one point", 1, MemberCard.pointsFor(10_001));
        check("a typical order", 6, MemberCard.pointsFor(63_000));
        try {
            MemberCard.pointsFor(-1);
            failed++;
            System.out.println("FAIL  a negative amount is refused: no exception");
        } catch (IllegalArgumentException e) {
            passed++;
            System.out.println("PASS  a negative amount is refused");
        }
        System.out.println(passed + " passed, " + failed + " failed");
        if (failed > 0) {
            System.exit(1);
        }
    }
}
```

The simplest code that passes is one integer division — “green” — and **Listing 6.55** is that code, documented. Nothing needed refactoring.

Listing 6.55 `MemberCard.java`: the loyalty-card rule, written to make the tests pass.

```
/**
 * Loyalty points for Profitina Laundry members.
 */
public class MemberCard {
    /** Rupiah needed for one point. */
    public static final long RUPIAH_PER_POINT = 10_000;

    /**
     * Returns the points earned for a payment: one point per full 10.000 rupiah.
     *
     * @param amount the amount paid, in rupiah
     * @return the number of points earned
     * @throws IllegalArgumentException if {@code amount} is negative
     */
    public static long pointsFor(long amount) {
        if (amount < 0) {
            throw new IllegalArgumentException("amount must not be negative: " + amount);
        }
        return amount / RUPIAH_PER_POINT;   // integer division drops the incomplete part
    }
}
```

```
PASS  nothing paid, no points
PASS  just below one point
PASS  exactly one point
PASS  just above one point
PASS  a typical order
PASS  a negative amount is refused
6 passed, 0 failed
```

In Python the test file (**Listing 6.56**) uses `parametrize` for the five boundary rows; run before `member_card.py` exists, pytest stops with `ModuleNotFoundError: No module named 'member_card'` — red. **Listing 6.57** is the one-line rule that turns it green.

Listing 6.56 `test_member_card.py`: the pytest version of the loyalty-card tests.

```python
"""The tests for member_card, written BEFORE member_card itself."""
import pytest

from member_card import points_for


@pytest.mark.parametrize("amount, points", [
    (0, 0),             # nothing paid
    (9_999, 0),         # just below one point
    (10_000, 1),        # exactly one point
    (10_001, 1),        # just above
    (63_000, 6),        # a typical order
])
def test_points(amount, points):
    assert points_for(amount) == points


def test_negative_amount_is_refused():
    with pytest.raises(ValueError):
        points_for(-1)
```

Listing 6.57 `member_card.py`: the rule, with floor division.

```python
"""Loyalty points for Profitina Laundry members."""

RUPIAH_PER_POINT = 10_000


def points_for(amount):
    """Return the points earned for a payment: one point per full 10.000 rupiah.

    Raises:
        ValueError: if amount is negative.
    """
    if amount < 0:
        raise ValueError(f"amount must not be negative: {amount}")
    return amount // RUPIAH_PER_POINT       # floor division drops the incomplete part
```

```
test_member_card.py::test_points[0-0] PASSED                              [ 16%]
test_member_card.py::test_points[9999-0] PASSED                           [ 33%]
test_member_card.py::test_points[10000-1] PASSED                          [ 50%]
test_member_card.py::test_points[10001-1] PASSED                          [ 66%]
test_member_card.py::test_points[63000-6] PASSED                          [ 83%]
test_member_card.py::test_negative_amount_is_refused PASSED               [100%]
============================== 6 passed in 0.01s ===============================
```

Solution 6.7. Equality now involves two fields, so both `equals` and `hashCode` use both (Equation (6.3)). `Objects.hash(number, brand)` combines the two hash codes; in Python, hashing the tuple `(number, brand)` does the same. **Listing 6.58** and **Listing 6.59** are the two classes with a small demonstration.

Listing 6.58 `Machine.java`: equality and hash code built from two fields.

```
import java.util.HashSet;
import java.util.Objects;
import java.util.Set;

/**
 * A washing machine, identified by its number and brand.
 */
public final class Machine {
    private final int number;
    private final String brand;

    public Machine(int number, String brand) {
        this.number = number;
        this.brand = brand;
    }

    @Override
    public boolean equals(Object other) {
        if (this == other) {
            return true;
        }
        if (!(other instanceof Machine)) {
            return false;
        }
        Machine that = (Machine) other;
        return number == that.number && brand.equals(that.brand);
    }

    @Override
    public int hashCode() {
        return Objects.hash(number, brand);   // uses exactly the fields equals uses
    }

    @Override
    public String toString() {
        return "Machine #" + number + " (" + brand + ")";
    }
```

Listing 6.58 (Continued) *Machine.java: equality and hash code built from two fields.*

```
    public static void main(String[] args) {
        Set<Machine> broken = new HashSet<>();
        broken.add(new Machine(3, "Tosaki"));
        broken.add(new Machine(3, "Tosaki"));   // reported twice by two staff members
        System.out.println("machines to repair: " + broken.size());
        System.out.println("is #3 broken? " + broken.contains(new Machine(3, "Tosaki")));
        System.out.println(broken);
    }
}
```

```
machines to repair: 1
is #3 broken? true
[Machine #3 (Tosaki)]
```

Listing 6.59 `machine.py`: `__eq__` and `__hash__` built from a tuple of two fields.

```
"""A washing machine, identified by its number and brand."""


class Machine:
    def __init__(self, number, brand):
        self._number = number
        self._brand = brand

    def __eq__(self, other):
        if not isinstance(other, Machine):
            return NotImplemented
        return (self._number, self._brand) == (other._number, other._brand)

    def __hash__(self):
        return hash((self._number, self._brand))   # exactly the fields __eq__ uses

    def __repr__(self):
        return f"Machine #{self._number} ({self._brand})"


broken = {Machine(3, "Tosaki"), Machine(3, "Tosaki")}   # reported twice
print("machines to repair:", len(broken))
print("is #3 broken?", Machine(3, "Tosaki") in broken)
print(broken)
```

```
machines to repair: 1
is #3 broken? True
{Machine #3 (Tosaki)}
```

Solution 6.8. **Listing 6.60** and **Listing 6.61** show that the generated methods compare *every* component. “Sari” and “Sari W.” differ in the name, so the set holds two records.

Listing 6.60 `CustomerRecordDemo.java`: a record compares all of its components.

```
import java.util.HashSet;
import java.util.Set;

/**
 * A record gets equals, hashCode and toString written by the compiler.
 */
public class CustomerRecordDemo {
    record CustomerRecord(String name, String phone) {
    }

    public static void main(String[] args) {
        CustomerRecord a = new CustomerRecord("Sari", "08125550101");
        CustomerRecord b = new CustomerRecord("Sari", "08125550101");
        CustomerRecord c = new CustomerRecord("Sari W.", "08125550101");
        System.out.println(a);
        System.out.println("a.equals(b): " + a.equals(b));
        System.out.println("a.equals(c): " + a.equals(c));   // every component counts
        Set<CustomerRecord> set = new HashSet<>();
        set.add(a);
        set.add(b);
        set.add(c);
        System.out.println("set size   : " + set.size());
    }
}
```

```
CustomerRecord[name=Sari, phone=08125550101]
a.equals(b): true
a.equals(c): false
set size   : 2
```

Listing 6.61 `customer_record.py`: a frozen dataclass compares all of its fields.

```python
"""A frozen dataclass gets __eq__, __hash__ and __repr__ written for it."""
from dataclasses import dataclass


@dataclass(frozen=True)
class CustomerRecord:
    name: str
    phone: str


a = CustomerRecord("Sari", "08125550101")
b = CustomerRecord("Sari", "08125550101")
c = CustomerRecord("Sari W.", "08125550101")
print(a)
print("a == b  :", a == b)
print("a == c  :", a == c)           # every field counts
print("set size:", len({a, b, c}))
```

```
CustomerRecord(name='Sari', phone='08125550101')
a == b  : True
a == c  : False
set size: 2
```

`Customer` in **Listing 6.15** decided that only the normalized phone identifies a customer, so the same two entries would count as one. Neither answer is "right" in general; the class decides what equality means, and a record or data class is the right tool only when the decision is "all fields".

Solution 6.9. **Listing 6.62** and **Listing 6.63** are the puzzles.

Listing 6.62 `OverloadPuzzle.java`: overload resolution with widening and boxing.

```java
/**
 * Which describe runs? Predict each line before you run the program.
 */
public class OverloadPuzzle {
    static String describe(int pieces) {
        return "int";
    }

    static String describe(double kilos) {
        return "double";
    }

    static String describe(Object anything) {
        return "Object";
    }

    public static void main(String[] args) {
        short s = 3;
        System.out.println("describe(s)                  : " + describe(s));
        System.out.println("describe(3f)                 : " + describe(3f));
        System.out.println("describe(Integer.valueOf(3)) : " +
describe(Integer.valueOf(3)));
        System.out.println("describe(\"3\")                : " + describe("3"));
        System.out.println("describe('3')                : " + describe('3'));
    }
}
```

```
describe(s)                  : int
describe(3f)                 : double
describe(Integer.valueOf(3)) : Object
describe("3")                : Object
describe('3')                : int
```

A `short` widens to `int`, and a `float` to `double`. `Integer.valueOf(3)` is already an object, and the first phase, which allows widening but not unboxing, finds `describe(Object)` at once. A `String` can go only to `Object`, and `'3'` is the character code 51, which widens to `int`.

Listing 6.63 `overload_puzzle.py`: `singledispatch` with `bool`, `Fraction` and a big integer.

```
"""Which describe runs? Predict each line before you run the program."""
from fractions import Fraction
from functools import singledispatch


@singledispatch
def describe(value):
    return "object"


@describe.register
def _(value: int):
    return "int"


@describe.register
def _(value: float):
    return "float"


print("describe(3)            :", describe(3))
print("describe(3.0)          :", describe(3.0))
print("describe(False)        :", describe(False))
print("describe('3')          :", describe("3"))
print("describe(Fraction(3,1)):", describe(Fraction(3, 1)))
print("describe(2 ** 100)     :", describe(2 ** 100))
```

```
describe(3)            : int
describe(3.0)          : float
describe(False)        : int
describe('3')          : object
describe(Fraction(3,1)): object
describe(2 ** 100)     : int
```

`False` is a `bool`, a subclass of `int`, so it takes the `int` version. `Fraction(3, 1)` equals 3 but is neither `int` nor `float`. And `2 ** 100` is still an `int`, because Python integers have no size limit. `singledispatch` looks only at the class of the first argument, never at its value.

Solution 6.10. The trick is an **adapter**: a small class that wraps an `Order` and presents it through the interface `average` understands. In Java (**Listing 6.64**) `LoadWeight` implements `Measurable` and returns the order's weight; `Order` and `average` stay exactly as they were.

Listing 6.64 `AverageLoad.java`: an adapter lets an `Order` take part in `average`.

```java
/**
 * Lets an Order take part in MeasurableDemo.average without changing Order.
 */
public class AverageLoad {
    /** An adapter: it wraps an Order and presents it as a Measurable. */
    static class LoadWeight implements Measurable {
        private final Order order;

        LoadWeight(Order order) {
            this.order = order;
        }

        @Override
        public double getMeasure() {
            return order.getWeightKg();
        }
    }

    public static void main(String[] args) {
        Measurable[] loads = {
            new LoadWeight(new Order("A001", "Sari", 2.0, 0, false)),
            new LoadWeight(new Order("A002", "Budi", 3.2, 0, true)),
            new LoadWeight(new Order("A005", "Sari", 5.5, 0, false))
        };
        System.out.printf("average load: %.2f kg%n", MeasurableDemo.average(loads));
    }
}
```

```
average load: 3.57 kg
```

By Equation (6.4), $(2.0 + 3.2 + 5.5)/3 = 10.7/3 \approx 3.57$. In Python (**Listing 6.65**) the adapter needs only a `get_measure` method, because `Measurable` is a protocol and fits by shape.

Listing 6.65 `average_load.py`: the same adapter in Python.

```python
"""Averaging order weights with the Measurable protocol: a small adapter is enough."""
from measurable_demo import average
from order import Order


class LoadWeight:
    """Wraps an Order and offers get_measure(), so it satisfies Measurable."""

    def __init__(self, order):
        self._order = order

    def get_measure(self):
        return self._order.weight_kg


loads = [LoadWeight(Order("A001", "Sari", 2.0, 0, False)),
         LoadWeight(Order("A002", "Budi", 3.2, 0, True)),
         LoadWeight(Order("A005", "Sari", 5.5, 0, False))]
print(f"average load: {average(loads):.2f} kg")
```

```
average load: 3.57 kg
```

The adapter is another pattern named in [15], useful whenever a class you should not change must fit an interface it was not written for.

Solution 6.11. The expected prices first, by Equation (6.6) and the discount written like Equation (6.8): 9 kg is not over the threshold, $9 \times 7\,000 = 63\,000$. Exactly 10 kg is not *strictly* heavier, so 70 000. 10.01 kg is charged as 11 kg, 77 000, and 20% off is $77\,000 - 15\,400 = 61\,600$. 12 kg is $84\,000 - 16\,800 = 67\,200$. **Listing 6.66** is the rule. Notice that it tests the *weight* of the

order, not the price, so it asks the order through its getter while the price comes from the wrapped rule. **Listing 6.67** checks the four boundaries and the refusal.

Listing 6.66 `BulkDiscountRule.java`: a decorator that discounts heavy orders.

```
/**
 * Gives a percentage off orders heavier than a threshold.
 */
public class BulkDiscountRule implements PriceRule {
    private final PriceRule base;
    private final double thresholdKg;
    private final int percent;

    /**
     * Wraps another rule.
     *
     * @param base        the rule that prices the order before the discount
     * @param thresholdKg orders strictly heavier than this get the discount
     * @param percent     the discount, from 0 to 100
     * @throws IllegalArgumentException if {@code percent} is outside 0..100
     */
    public BulkDiscountRule(PriceRule base, double thresholdKg, int percent) {
        if (percent < 0 || percent > 100) {
            throw new IllegalArgumentException("percent must be 0..100: " + percent);
        }
        this.base = base;
        this.thresholdKg = thresholdKg;
        this.percent = percent;
    }

    @Override
    public long priceOf(Order order) {
        long normal = base.priceOf(order);
        if (order.getWeightKg() <= thresholdKg) {
            return normal;
        }
        return normal - normal * percent / 100;
    }

    @Override
    public String name() {
        return base.name() + " + bulk " + percent + "%";
    }
}
```

Listing 6.67 `BulkDiscountCheck.java`: the boundary tests for the bulk discount.

```
/**
 * Boundary tests for BulkDiscountRule.
 */
public class BulkDiscountCheck {
    private static int passed = 0;
    private static int failed = 0;

    static void check(String what, long expected, long actual) {
        if (expected == actual) {
            passed++;
            System.out.println("PASS  " + what);
        } else {
            failed++;
            System.out.println("FAIL  " + what + ": expected " + expected + " but was " +
actual);
        }
    }

    static Order kg(double weight) {
        return new Order("B1", "Test", weight, 0, false);
    }

    public static void main(String[] args) {
        PriceRule bulk = new BulkDiscountRule(new PerKiloRule("wash & fold", 7_000, 3), 10,
20);
        check("9 kg: no discount", 63_000, bulk.priceOf(kg(9)));
        check("exactly 10 kg: no discount", 70_000, bulk.priceOf(kg(10)));
        check("10.01 kg: 11 kg charged, 20% off", 61_600, bulk.priceOf(kg(10.01)));
        check("12 kg: 20% off", 67_200, bulk.priceOf(kg(12)));
        try {
            new BulkDiscountRule(new FlatRule("flat", 1), 10, -5);
            failed++;
            System.out.println("FAIL  a negative percent is refused: no exception");
        } catch (IllegalArgumentException e) {
            passed++;
            System.out.println("PASS  a negative percent is refused");
        }
        System.out.println(passed + " passed, " + failed + " failed");
        if (failed > 0) {
            System.exit(1);
        }
    }
}
```

```
PASS  9 kg: no discount
PASS  exactly 10 kg: no discount
PASS  10.01 kg: 11 kg charged, 20% off
PASS  12 kg: 20% off
PASS  a negative percent is refused
5 passed, 0 failed
```

The Python rule (**Listing 6.68**) is a subclass of the ABC `PriceRule`, and its tests (**Listing 6.69**) are one parametrized function.

Listing 6.68 `bulk_discount.py`: the bulk discount in Python.

```
"""A price rule that gives a percentage off orders heavier than a threshold."""
from price_rule import PriceRule


class BulkDiscountRule(PriceRule):
    """Orders strictly heavier than threshold_kg get percent off."""

    def __init__(self, base, threshold_kg, percent):
        if not 0 <= percent <= 100:
            raise ValueError(f"percent must be 0..100: {percent}")
        self._base = base
        self._threshold_kg = threshold_kg
        self._percent = percent

    def price_of(self, order):
        normal = self._base.price_of(order)
        if order.weight_kg <= self._threshold_kg:
            return normal
        return normal - normal * self._percent // 100

    def name(self):
        return f"{self._base.name()} + bulk {self._percent}%"
```

Listing 6.69 `test_bulk_discount.py`: pytest boundary tests for the bulk discount.

```
"""Boundary tests for BulkDiscountRule."""
import pytest

from bulk_discount import BulkDiscountRule
from order import Order
from rules import FlatRule, PerKiloRule

BULK = BulkDiscountRule(PerKiloRule("wash & fold", 7_000, 3), 10, 20)


@pytest.mark.parametrize("kg, expected", [
    (9, 63_000),         # below the threshold
    (10, 70_000),        # exactly on it: no discount
    (10.01, 61_600),     # just past it: 11 kg charged, 20% off
    (12, 67_200),
])
def test_bulk_boundaries(kg, expected):
    assert BULK.price_of(Order("B1", "Test", kg, 0, False)) == expected


def test_negative_percent_is_refused():
    with pytest.raises(ValueError):
        BulkDiscountRule(FlatRule("flat", 1), 10, -5)
```

```
test_bulk_discount.py::test_bulk_boundaries[9-63000] PASSED              [ 20%]
test_bulk_discount.py::test_bulk_boundaries[10-70000] PASSED             [ 40%]
test_bulk_discount.py::test_bulk_boundaries[10.01-61600] PASSED          [ 60%]
test_bulk_discount.py::test_bulk_boundaries[12-67200] PASSED             [ 80%]
test_bulk_discount.py::test_negative_percent_is_refused PASSED           [100%]
============================== 5 passed in 0.01s ===============================
```

Nothing else changed — not `Order`, not `PriceRule`, not the other rules or their tests — which is Section 6.16.7 in practice.

Solution 6.12. Listing 6.70 tries the 21 bases.

Listing 6.70 `DecoratorOrder.java`: stacking the same two decorators in both orders.

```java
/**
 * Does the order in which ExpressRule and MemberDiscountRule are stacked matter?
 */
public class DecoratorOrder {
    public static void main(String[] args) {
        Order member = new Order("D1", "Test", 0, 1, true);
        int differences = 0;
        for (long base = 21_000; base <= 21_020; base++) {
            PriceRule flat = new FlatRule("flat", base);
            long expressFirst = new MemberDiscountRule(new ExpressRule(flat),
10).priceOf(member);
            long discountFirst = new ExpressRule(new MemberDiscountRule(flat,
10)).priceOf(member);
            if (expressFirst != discountFirst) {
                differences++;
                System.out.println("base " + base + ": express first " + expressFirst
                        + ", discount first " + discountFirst);
            }
        }
        System.out.println(differences + " of 21 bases differ");
    }
}
```

```
base 21007: express first 28359, discount first 28360
base 21008: express first 28361, discount first 28362
base 21009: express first 28362, discount first 28363
base 21010: express first 28364, discount first 28363
base 21012: express first 28367, discount first 28366
base 21015: express first 28370, discount first 28371
base 21017: express first 28373, discount first 28374
base 21019: express first 28376, discount first 28377
8 of 21 bases differ
```

The Python program, **Listing 6.71**, prints the same eight lines.

Listing 6.71 `decorator_order.py`: the same experiment in Python.

```python
"""Does the order in which ExpressRule and MemberDiscountRule are stacked matter?"""
from order import Order
from rules import ExpressRule, FlatRule, MemberDiscountRule

member = Order("D1", "Test", 0, 1, True)
differences = 0
for base in range(21_000, 21_021):
    flat = FlatRule("flat", base)
    express_first = MemberDiscountRule(ExpressRule(flat), 10).price_of(member)
    discount_first = ExpressRule(MemberDiscountRule(flat, 10)).price_of(member)
    if express_first != discount_first:
        differences += 1
        print(f"base {base}: express first {express_first}, discount first
{discount_first}")
print(differences, "of 21 bases differ")
```

The two orders differ in 8 of 21 cases, always by one rupiah, because each decorator rounds down *at its own step*. Take base 21 007. Express first: $21\,007 + 10\,503 = 31\,510$, then minus $\lfloor 3\,151.0 \rfloor = 3\,151$, giving 28 359. Discount first: $21\,007 - \lfloor 2\,100.7 \rfloor = 18\,907$, then plus $\lfloor 9\,453.5 \rfloor = 9\,453$, giving 28 360. Mathematically the operations commute; with rounding at every step they do not. The laundry should *decide* one order, write it into the documentation of the factory `ruleFor`, and add a test with an awkward base such as 21 007 so that nobody swaps the order unnoticed. (`ruleFor` in **Listing 6.42** applies express first, then the member discount.)

Solution 6.13. Listing 6.72 implements `Comparator<Order>`. `Double.compare(b, a)` with the arguments reversed gives the descending order by weight; when the weights are equal the ticket

ids decide, using `String.compareTo` — whose sign, not value, is all we pass on, as Equation (6.5) requires.

Listing 6.72 `OrderSortDemo.java`: a `Comparator` that sorts heaviest first, then by ticket id.

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

/**
 * Sorting orders with a Comparator: another interface from java.util.
 */
public class OrderSortDemo {
    /** Heaviest order first; equal weights in ticket-id order. */
    static class HeaviestFirst implements Comparator<Order> {
        @Override
        public int compare(Order a, Order b) {
            int byWeight = Double.compare(b.getWeightKg(), a.getWeightKg());  // b before
a: descending
            if (byWeight != 0) {
                return byWeight;
            }
            return a.getId().compareTo(b.getId());
        }
    }

    public static void main(String[] args) {
        List<Order> orders = new ArrayList<>(List.of(
                new Order("A003", "Dewi", 3.2, 0, true),
                new Order("A001", "Sari", 2.0, 0, false),
                new Order("A005", "Sari", 5.5, 0, false),
                new Order("A002", "Budi", 3.2, 0, true)));
        orders.sort(new HeaviestFirst());
        for (Order o : orders) {
            System.out.println(o);
        }
    }
}
```

```
Order[A005, Sari, 5.5 kg, 0 pcs]
Order[A002, Budi, 3.2 kg, 0 pcs, member]
Order[A003, Dewi, 3.2 kg, 0 pcs, member]
Order[A001, Sari, 2.0 kg, 0 pcs]
```

In Python (**Listing 6.73**) the key `(-o.weight_kg, o.id)` says the same thing as a tuple: the minus sign reverses the order of weights, and the id breaks ties.

Listing 6.73 `order_sort_demo.py`: sorting with a tuple key.

```
"""Sorting orders with a key function: heaviest first, then by ticket id."""
from order import Order

orders = [Order("A003", "Dewi", 3.2, 0, True),
          Order("A001", "Sari", 2.0, 0, False),
          Order("A005", "Sari", 5.5, 0, False),
          Order("A002", "Budi", 3.2, 0, True)]
for o in sorted(orders, key=lambda o: (-o.weight_kg, o.id)):   # minus: descending
    print(o)
```

```
Order[A005, Sari, 5.5 kg, 0 pcs]
Order[A002, Budi, 3.2 kg, 0 pcs, member]
Order[A003, Dewi, 3.2 kg, 0 pcs, member]
Order[A001, Sari, 2.0 kg, 0 pcs]
```

`Comparable` describes an object's single *natural* order. Orders have none: the counter wants them by ticket id, the washing room heaviest first, the courier by address. A `Comparator` is an ordering chosen from outside, and a class can have as many as its users need.

References

[1] D. Goodger and G. van Rossum, "PEP 257 – Docstring Conventions," Python Software Foundation, 2001. [Online]. Available: https://peps.python.org/pep-0257/

[2] Python Software Foundation, "doctest — Test interactive Python examples," *The Python Standard Library*, 2025. [Online]. Available: https://docs.python.org/3/library/doctest.html

[3] Oracle, "The javadoc Command," *Java Development Kit Version 21 Tool Specifications*, 2023. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/specs/man/javadoc.html

[4] The JUnit Team, *JUnit User Guide*, version 6.1.3, Aug. 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://docs.junit.org/current/

[5] Python Software Foundation, "unittest — Unit testing framework," *The Python Standard Library*, 2025. [Online]. Available: https://docs.python.org/3/library/unittest.html

[6] H. Krekel and the pytest-dev team, *pytest Documentation*, 2025. [Online]. Available: https://docs.pytest.org/

[7] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Hoboken, NJ, USA: Wiley, 2011.

[8] K. Beck, *Test-Driven Development: By Example*. Boston, MA, USA: Addison-Wesley, 2002.

[9] Oracle, "Class Object," *Java Platform, Standard Edition 21 API Specification*, 2023. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Object.html

[10] J. Bloch, *Effective Java*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2018.

[11] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*, §15.12.2 "Compile-Time Step 2: Determine Method Signature," Oracle, 2023. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se21/html/

[12] Python Software Foundation, "functools — Higher-order functions and operations on callable objects," *The Python Standard Library*, 2025. [Online]. Available: https://docs.python.org/3/library/functools.html

[13] Python Software Foundation, "abc — Abstract Base Classes," *The Python Standard Library*, 2025. [Online]. Available: https://docs.python.org/3/library/abc.html

[14] I. Levkivskyi, J. Lehtosalo, and Ł. Langa, "PEP 544 – Protocols: Structural subtyping (static duck typing)," Python Software Foundation, 2017. [Online]. Available: https://peps.python.org/pep-0544/

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

Chapter 7

Inheritance, Polymorphism, and Access Control

By the end of this chapter you will be able to spot the kind of duplicated code that inheritance was invented to remove, and remove it; write a superclass and its subclasses in Java with `extends` and in Python with `class Sub(Base)`; chain constructors correctly with `super(...)` and `super().__init__(...)`; redefine (override) an inherited method and tell overriding apart from overloading at a glance; explain, with a picture of memory, exactly how a single line such as `s.price()` can run a different method for every object it meets; decide when a class should be abstract; choose between inheritance and composition and defend the choice; read Python's method resolution order and compare it with Java's single inheritance plus interfaces; and use `public`, `protected`, package access and `private` in Java, and the `_name`, `__name` and `@property` conventions in Python, to build classes that protect their own data. All of it comes together in a working price engine for Profitina Laundry, in which one loop prices washing, ironing, dry cleaning and express orders without knowing which is which.

7.1 Where We Left Off

Chapter 3 gave Profitina Laundry its first real class, `Service`, with a code, a name, a price per kilogram and a minimum weight. It worked. It also had a small smell that we did not stop to discuss: to answer "how long will this take?", the class looked at its own code with a `switch` — `"WF"` meant two days, `"IR"` one day, and so on — and whether an order was *express* was a `boolean` passed in from outside, tested with an `if` inside the pricing method. Chapter 5 put many services and orders into lists and taught us to fail loudly with exceptions. Chapter 6 taught us to document and test a class, showed us how an *interface* states what a class promises without saying how, and introduced *overloading*: several methods with the same name and different parameter lists.

Now imagine the laundry grows. The owner adds dry cleaning, which is priced per piece rather than per kilogram. Then shoe cleaning, priced per pair. Then a promotion that takes 20% off one service. Every new kind of service means opening `Service` again, adding another `case` to the `switch`, another `if` to the price method, and hoping nothing that used to work has broken. After a year the class is a thousand lines of special cases that nobody dares to touch.

This chapter is about the three ideas that object-oriented programming offers against that decay. **Inheritance** lets us write the common part of several classes once, in a *superclass*, and let each *subclass* add or change only what is different about it. **Polymorphism** lets one line of code, such as `s.price()`, do the right thing for every kind of object it is given, so the code that *uses* services never needs a `switch` at all. **Access control** lets each class decide which of its parts the rest of the program may touch, so that the promises a class makes cannot be broken from outside.

We meet the ideas in the order a working programmer does: a problem (two classes that repeat each other), its cure (a superclass), the surprises the cure brings (which method runs? what happens to constructors?), and the discipline that keeps a hierarchy healthy (abstract classes,

composition, interfaces, access modifiers). Every idea is shown in Java, then in Python, then run, then explained.

> ✓ **KEY INSIGHT.** Inheritance is about *not writing the same code twice*. Polymorphism is about *not asking what kind of object you have*. Access control is about *not letting anyone break an object's promises*. Keep these three sentences in mind; everything in this chapter is a detail of one of them.

7.2 The Duplication Problem: Two Databases That Repeat Each Other

We begin not with laundry but with a small, clean example that shows the problem in its purest form. Suppose we are writing a tiny database that stores integer keys. We need two implementations, because they are good at different things:

- `LinearDB` keeps the keys in an array, in the order they arrive. Adding a key is instant (put it at the end); finding a key means walking the array from the start.
- `TreeDB` keeps the keys in a binary search tree. Adding and finding both walk one path from the root down, which is short when the tree is balanced.

Both classes need the same four operations:

```
void    addKey(int key)          add one key
boolean search(int key)          is this key stored?
void    addSeveral(int keys[])   add every key in the array
boolean findOneOf(int keys[])    is at least one of these keys stored?
```

Look closely at the last two. How do you add several keys? You call `addKey` once for each of them. How do you find whether one of several keys is present? You call `search` for each key and stop at the first success. Neither answer mentions arrays or trees. `addSeveral` and `findOneOf` do not care *how* a key is stored; they only need *some* `addKey` and *some* `search` to call.

So if we write `LinearDB` and `TreeDB` as two unrelated classes, we write `addSeveral` and `findOneOf` twice, letter for letter. The left half of **Figure 7.1** shows the result: two classes, eight methods, four of them identical copies. Duplicated code is not merely wasteful. It is a trap. The day someone finds a bug in `findOneOf` and fixes it in `LinearDB`, the copy in `TreeDB` keeps the bug, and the two classes that were meant to behave alike now quietly disagree.

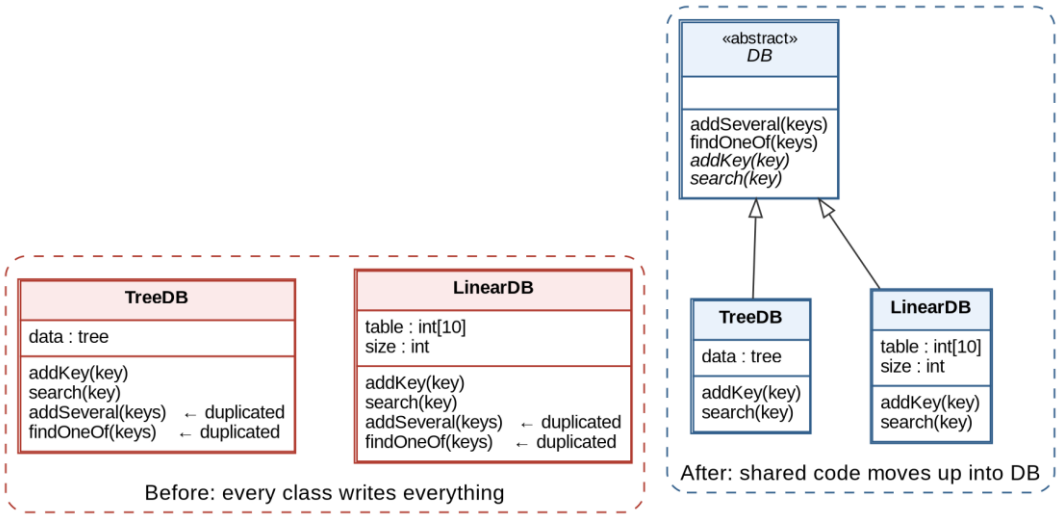


Figure 7.1 The duplication problem and its cure. Left: two independent classes each carry their own copy of `addSeveral` and `findOneOf`. Right: the shared methods move up into the abstract superclass `DB`; each subclass keeps only what is genuinely its own.

The right half of **Figure 7.1** shows the cure. We invent a third class, `DB`, that holds everything the two databases have in common: the two methods whose code is the same, plus a *declaration* — a name and a parameter list, with no body — of the two methods whose code differs. `LinearDB` and `TreeDB` then say "I am a kind of `DB`", and they automatically receive `addSeveral` and `findOneOf`. Each writes only its own `addKey` and `search`.

This idea is older than Java and Python by thirty years. It first appeared in Simula 67, a Norwegian language designed for simulating systems of interacting objects, whose designers noticed that different kinds of simulated objects shared most of their behavior and wanted to write that behavior once [1]. Every mainstream object-oriented language since has inherited (the pun is irresistible) the same mechanism.

7.2.1 The vocabulary

> **DEFINITION.** When class `C` is declared as an extension of class `B`, we say that `C` **inherits from** `B`. `B` is the **superclass** (also *base class* or *parent class*) and `C` is the **subclass** (also *derived class* or *child class*). The subclass automatically has every field and every method of the superclass, in addition to those it declares itself.

In our example `DB` is the superclass, and `LinearDB` and `TreeDB` are its subclasses. We say that `LinearDB` and `TreeDB` *inherit from* `DB`, because they obtain the definitions of `addSeveral` and `findOneOf` from it without writing them.

The relationship has a name that you should say aloud whenever you design a hierarchy: **is-a**. A `LinearDB` *is a* `DB`. A `TreeDB` *is a* `DB`. If the sentence sounds wrong — "a `Customer` is a `Service`" — the inheritance is wrong, however much code it would save. We return to this test in Section 7.11.

7.3 Superclasses and Subclasses in Java and Python

Listing 7.1 is the database project written in Java. Everything lives in one file, `DBTest.java`, in a package called `db` (a *package* is simply a named folder of classes; Section 7.13 explains why packages matter for access control). The file holds four classes: the public class `DBTest`, which contains `main`, and three helper classes that are visible only inside the package.

Listing 7.1 `db/DBTest.java`: the database project with an abstract superclass `DB` and two subclasses, `LinearDB` and `TreeDB`, whose methods only report that they were called.

```
package db;

public class DBTest {
    public static void main(String[] args) {
        DB ldb = new LinearDB();
        DB tdb = new TreeDB();
        ldb.addKey(1);
        if (ldb.search(1)) {
            System.out.println("It's found!");
        } else {
            System.out.println("It's not found!");
        }
        tdb.addKey(2);
        if (tdb.search(1)) {
            System.out.println("It's found!");
        } else {
            System.out.println("It's not found!");
        }
        // The two methods below are written once, in DB, and inherited.
        System.out.println("-- inherited addSeveral and findOneOf --");
        ldb.addSeveral(new int[] {3, 4});
        System.out.println("findOneOf: " + tdb.findOneOf(new int[] {5, 6}));
    }
}

abstract class DB {
    void addSeveral(int keys[]) {
        for (int i = 0; i < keys.length; i++) {
            addKey(keys[i]);
        }
    }

    boolean findOneOf(int keys[]) {
        for (int i = 0; i < keys.length; i++) {
            if (search(keys[i])) {
                return true;
            }
        }
        return false;
    }

    abstract void addKey(int key);     // No code
    abstract boolean search(int key);  // No code
}
```

Listing 7.1 (Continued) *db/DBTest.java: the database project with an abstract superclass DB and two subclasses, LinearDB and TreeDB, whose methods only report that they were called.*

```
class LinearDB extends DB {
    int table[] = new int[10];
    int size = 0;

    void addKey(int key) {
        System.out.println("LinearDB: addKey()");
    }

    boolean search(int key) {
        boolean result = true;
        System.out.println("LinearDB: searchKey()");
        return result;
    }
}

class TreeDB extends DB {
    Object data;  // a binary search tree would live here

    void addKey(int key) {
        System.out.println("TreeDB: addKey()");
    }

    boolean search(int key) {
        boolean result = false;
        System.out.println("TreeDB: searchKey()");
        return result;
    }
}
```

To compile and run it, open a terminal in the chapter's `java` folder and type `javac -d out db/DBTest.java` and then `java -cp out db.DBTest` (the README of the code folder repeats this for Windows, macOS and Linux). The program prints:

```
LinearDB: addKey()
LinearDB: searchKey()
It's found!
TreeDB: addKey()
TreeDB: searchKey()
It's not found!
-- inherited addSeveral and findOneOf --
LinearDB: addKey()
LinearDB: addKey()
TreeDB: searchKey()
TreeDB: searchKey()
findOneOf: false
```

Let us read the listing line by line, starting with the superclass.

- `abstract class DB {` begins the superclass. The word `abstract` means "this class is incomplete; you may not create objects of it directly". It is incomplete because two of its methods have no body, as we see in a moment. Section 7.9 is devoted to abstract classes; for now accept that `DB` exists only to be inherited from.
- `void addSeveral(int keys[])` is an ordinary method with an ordinary body: a `for` loop that calls `addKey(keys[i])` for each key. Notice what it calls. `DB` has no code for `addKey` — yet the call compiles, because `DB` *declares* that every `DB` has an `addKey`.
- `boolean findOneOf(int keys[])` is the same idea: loop, call `search`, return `true` at the first success, `false` if none succeeds.

- `abstract void addKey(int key);` and `abstract boolean search(int key);` are *abstract methods*: a return type, a name and a parameter list, then a semicolon instead of a body. They are promises: "every concrete kind of `DB` will supply this method".
- `class LinearDB extends DB {` is the heart of the listing. The keyword `extends` makes `LinearDB` a subclass of `DB`. `LinearDB` then declares two fields of its own, `table` and `size`, and supplies bodies for `addKey` and `search`. In this first version the bodies merely print a line and return a fixed answer; `search` always says `true`. That is deliberate: the program is a skeleton that shows *which* method runs, not a working database. A working one follows in Section 7.4.
- `class TreeDB extends DB {` does the same for the tree version. Its `search` always says `false`.
- In `main`, the line `DB ldb = new LinearDB();` deserves a slow reading. The *variable* `ldb` is declared with type `DB`. The *object* created is a `LinearDB`. This is allowed precisely because a `LinearDB` is a `DB`: wherever the program expects a `DB`, any subclass object will do. We cannot, however, write `new DB()`, because `DB` is abstract; there are no `DB` objects, only `LinearDB` and `TreeDB` objects.

Now match the output to the code. `ldb.addKey(1)` printed `LinearDB: addKey()`: the call went to `LinearDB`'s method even though the variable's type is `DB`. `ldb.search(1)` returned `true`, so the program printed `It's found!`; the tree returned `false`, hence `It's not found!`. The two extra lines at the end of `main` exercise the inherited methods. `ldb.addSeveral(new int[] {3, 4})` runs the loop written in `DB`, and the output shows `LinearDB: addKey()` twice: the inherited loop called `LinearDB`'s version. Likewise `tdb.findOneOf(new int[] {5, 6})` runs `DB`'s loop, which calls `TreeDB`'s `search` twice and so returns `false`.

That last observation is the whole of polymorphism in miniature, and we will spend Section 7.7 on it: **code written in the superclass calls methods that are chosen by the actual object's class**.

7.3.1 "It is as if LinearDB were defined as…"

A good way to understand what `extends` does is to imagine the compiler copying the superclass's members into the subclass. After inheritance, `LinearDB` behaves exactly as if we had written it like this:

```
class LinearDB {
    int table[] = new int[10];      // declared in LinearDB
    int size = 0;                   // declared in LinearDB
    void addKey(int key)            // declared in LinearDB
    boolean search(int key)         // declared in LinearDB
    void addSeveral(int keys[])     // inherited from DB
    boolean findOneOf(int keys[])   // inherited from DB
}
```

and similarly for `TreeDB`. The picture is a mental model, not what the machine literally does — no code is copied; Section 7.7 shows what really happens — but it predicts correctly which members a subclass has.

The rule extends down a chain. Given

```
abstract class B { … }
abstract class C extends B { … }
class D extends C { … }
```

`C` has all the fields and methods of `B` in addition to its own, and `D` has those of `C` — which already include those of `B` — in addition to its own. Inheritance is *transitive*: `D` inherits from both `C` and `B`. We will build a three-level chain for the laundry in Section 7.15.

7.3.2 The same program in Python

Listing 7.2 is the same project in Python. Python has no `abstract` keyword; instead the standard-library module `abc` (Abstract Base Classes) provides a base class `ABC` and a decorator `@abstractmethod`, and together they give Python the same "incomplete class" idea [2].

Listing 7.2 `db_test.py`: the database project in Python. `DB` inherits from `ABC` and marks `add_key` and `search` as abstract.

```
"""db_test.py: the database project, with DB as an abstract superclass."""
from abc import ABC, abstractmethod


class DB(ABC):
    def add_several(self, keys):
        for key in keys:
            self.add_key(key)

    def find_one_of(self, keys):
        for key in keys:
            if self.search(key):
                return True
        return False

    @abstractmethod
    def add_key(self, key):  # No code
        ...

    @abstractmethod
    def search(self, key):  # No code
        ...


class LinearDB(DB):
    def __init__(self):
        self.table = [0] * 10
        self.size = 0

    def add_key(self, key):
        print("LinearDB: addKey()")

    def search(self, key):
        result = True
        print("LinearDB: searchKey()")
        return result


class TreeDB(DB):
    def __init__(self):
        self.data = None  # a binary search tree would live here

    def add_key(self, key):
        print("TreeDB: addKey()")
```

Listing 7.2 (Continued) *db_test.py: the database project in Python. DB inherits from ABC and marks add_key and search as abstract.*

```
    def search(self, key):
        result = False
        print("TreeDB: searchKey()")
        return result


def main():
    ldb = LinearDB()
    tdb = TreeDB()
    ldb.add_key(1)
    if ldb.search(1):
        print("It's found!")
    else:
        print("It's not found!")
    tdb.add_key(2)
    if tdb.search(1):
        print("It's found!")
    else:
        print("It's not found!")
    # The two methods below are written once, in DB, and inherited.
    print("-- inherited add_several and find_one_of --")
    ldb.add_several([3, 4])
    print("find_one_of:", tdb.find_one_of([5, 6]))
    try:
        DB()
    except TypeError as e:
        print("DB():", e)


if __name__ == "__main__":
    main()
```

Run it with `python3 db_test.py` (on Windows, `python db_test.py` or `py db_test.py`):

```
LinearDB: addKey()
LinearDB: searchKey()
It's found!
TreeDB: addKey()
TreeDB: searchKey()
It's not found!
-- inherited add_several and find_one_of --
LinearDB: addKey()
LinearDB: addKey()
TreeDB: searchKey()
TreeDB: searchKey()
find_one_of: False
DB(): Can't instantiate abstract class DB with abstract methods add_key, search
```

The differences from Java are few and worth listing:

- **Inheritance syntax.** Python writes the superclass in parentheses after the class name: `class LinearDB(DB):`. There is no `extends`.
- **Abstract methods.** `DB` itself inherits from `ABC`, and each unfinished method carries `@abstractmethod`. The body `...` (the *Ellipsis*, Python's "nothing here yet") is a placeholder; it is never run.
- **Fields.** Python has no field declarations. An object's data is created by assigning to `self.something`, usually in `__init__`, the method Python calls to initialize a new object. So `LinearDB.__init__` creates `self.table` and `self.size`.
- **Names.** Python style writes methods in `snake_case` (`add_key`), Java in `camelCase` (`addKey`). The printed strings are kept identical to Java's so that the two outputs can be compared line by line.

- **The abstract class refuses to be created.** The last lines of `main` try `DB()` and catch the resulting `TypeError`. Java refuses at *compile time* (the program never runs); Python refuses at *run time*, the moment you call the constructor. The wording of the message changed in Python 3.12, which says `Can't instantiate abstract class DB without an implementation for abstract methods 'add_key', 'search'`; the meaning is identical.

For comparison, here is what Java says if you add the line `DB nothing = new DB();` right after `DB tdb = new TreeDB();` in **Listing 7.1**:

```
db/DBTest.java:7: error: DB is abstract; cannot be instantiated
        DB nothing = new DB();
                     ^
1 error
```

7.4 Redefining Methods: Overriding

A subclass that could only *add* to its superclass would be of limited use. The really important aspect of inheritance is that a subclass can also **redefine** an inherited method — supply its own body for a method the superclass already has. The precise rule is:

> **DEFINITION.** A subclass `C` of `B` inherits the instance variables and instance methods of `B`, **except those that `C` defines itself**. When `C` defines a method with the same name and the same parameter types as a method of `B`, we say that `C` **overrides** that method. Calls on a `C` object then run `C`'s version.

Why would `LinearDB` want to override `addSeveral`? Because the inherited version calls `addKey` once per key, and each call checks whether the array has room. An array-based database can do better: make room once, then copy all the keys in one block. A direct version for the skeleton of **Listing 7.1** could read:

```
void addSeveral(int keys[]) {
    for (int i = 0; i < keys.length; i++) {
        table[size + i] = keys[i];
    }
    size += keys.length;
}
```

That is faster, and it is legal because `LinearDB` knows about `table` and `size` while `DB` does not. It is also a little dangerous: it forgets to check that ten slots are enough. Let us write the database properly, with an array that grows and a real binary search tree, so that we can see overriding in a program that actually stores keys.

Listing 7.3 renames the classes `IntDB`, `LinearIntDB` and `TreeIntDB` (so that they can live in the same folder as the other programs of this chapter) and makes them real.

Listing 7.3 `IntDBDemo.java`: a working key database. `LinearIntDB` overrides the inherited `addSeveral` with a bulk copy; `TreeIntDB` inherits it unchanged.

```
import java.util.Arrays;

public class IntDBDemo {
    public static void main(String[] args) {
        IntDB[] dbs = {new LinearIntDB(), new TreeIntDB()};
        for (IntDB db : dbs) {
            System.out.println(db.name());
            db.addSeveral(new int[] {42, 7, 19, 3, 25});
            System.out.println("  size                = " + db.size());
            System.out.println("  search(19)          = " + db.search(19));
            System.out.println("  search(8)           = " + db.search(8));
            System.out.println("  findOneOf({8,9,25}) = " + db.findOneOf(new int[] {8, 9,
25}));
            System.out.println("  findOneOf({1,2})    = " + db.findOneOf(new int[] {1,
2}));
            System.out.println("  comparisons so far  = " + db.comparisons);
        }
    }
}

abstract class IntDB {
    protected int comparisons = 0;  // counted by search(), so we can compare the two

    // Written once, inherited by every subclass.
    void addSeveral(int[] keys) {
        for (int key : keys) {
            addKey(key);
        }
    }

    // Written once, inherited by every subclass.
    boolean findOneOf(int[] keys) {
        for (int key : keys) {
            if (search(key)) {
                return true;
            }
        }
        return false;
    }

    abstract void addKey(int key);
    abstract boolean search(int key);
    abstract int size();
    abstract String name();
}
```

Listing 7.3 (Continued) *IntDBDemo.java: a working key database. LinearIntDB overrides the inherited addSeveral with a bulk copy; TreeIntDB inherits it unchanged.*

```
class LinearIntDB extends IntDB {
    private int[] table = new int[4];
    private int size = 0;

    @Override
    void addKey(int key) {
        if (size == table.length) {
            table = Arrays.copyOf(table, 2 * table.length);  // grow when full
        }
        table[size++] = key;
    }

    // Redefinition: one bulk copy instead of one addKey() call per key.
    @Override
    void addSeveral(int[] keys) {
        while (size + keys.length > table.length) {
            table = Arrays.copyOf(table, 2 * table.length);
        }
        System.arraycopy(keys, 0, table, size, keys.length);
        size += keys.length;
        System.out.println("  (bulk copy of " + keys.length + " keys)");
    }

    @Override
    boolean search(int key) {
        for (int i = 0; i < size; i++) {
            comparisons++;
            if (table[i] == key) {
                return true;
            }
        }
        return false;
    }

    @Override
    int size() {
        return size;
    }

    @Override
    String name() {
        return "LinearIntDB (unsorted array)";
    }
}
```

Listing 7.3 (Continued) *IntDBDemo.java: a working key database. LinearIntDB overrides the inherited addSeveral with a bulk copy; TreeIntDB inherits it unchanged.*

```
class TreeIntDB extends IntDB {
    private static class Node {
        int key;
        Node left, right;

        Node(int key) {
            this.key = key;
        }
    }

    private Node root;
    private int size = 0;

    @Override
    void addKey(int key) {
        if (root == null) {
            root = new Node(key);
            size++;
            return;
        }
        Node cur = root;
        while (true) {
            if (key == cur.key) {
                return;  // already present: a set keeps one copy
            } else if (key < cur.key) {
                if (cur.left == null) {
                    cur.left = new Node(key);
                    size++;
                    return;
                }
                cur = cur.left;
            } else {
                if (cur.right == null) {
                    cur.right = new Node(key);
                    size++;
                    return;
                }
                cur = cur.right;
            }
        }
    }
```

Listing 7.3 (Continued) *IntDBDemo.java: a working key database. LinearIntDB overrides the inherited addSeveral with a bulk copy; TreeIntDB inherits it unchanged.*

```
    @Override
    boolean search(int key) {
        Node cur = root;
        while (cur != null) {
            comparisons++;
            if (key == cur.key) {
                return true;
            }
            cur = (key < cur.key) ? cur.left : cur.right;
        }
        return false;
    }

    @Override
    int size() {
        return size;
    }

    @Override
    String name() {
        return "TreeIntDB (binary search tree)";
    }
}
```

```
LinearIntDB (unsorted array)
  (bulk copy of 5 keys)
  size                = 5
  search(19)          = true
  search(8)           = false
  findOneOf({8,9,25}) = true
  findOneOf({1,2})    = false
  comparisons so far  = 33
TreeIntDB (binary search tree)
  size                = 5
  search(19)          = true
  search(8)           = false
  findOneOf({8,9,25}) = true
  findOneOf({1,2})    = false
  comparisons so far  = 22
```

Walk through the pieces.

- **`IntDB`** is the abstract superclass. It has one field, `comparisons`, marked `protected` — visible to subclasses (Section 7.13 gives the exact rule). Both subclasses increase it inside `search`, so that we can compare how hard the two data structures work. `addSeveral` and `findOneOf` are written once. `addKey`, `search`, `size` and `name` are abstract.
- **`LinearIntDB`** stores keys in `table`, starting with four slots. Its `addKey` checks whether the array is full and, if so, replaces it with a copy twice as large (`Arrays.copyOf(table, 2 * table.length)`); then it stores the key and increases `size`. Its `search` walks the array and counts one comparison per element examined.
- **The override.** `LinearIntDB` also defines `addSeveral(int[] keys)`, with exactly the parameter list of the inherited method. Its body doubles the array until everything fits, copies all the keys with one call to `System.arraycopy`, and prints a note so that we can see it ran. The annotation `@Override` above it tells the compiler "I intend this to override a superclass method; please check that it does". It changes nothing when the method is correct, and turns a silent mistake — a misspelled name, a wrong parameter type — into a compile error. Always write it.
- **`TreeIntDB`** stores keys in a binary search tree made of `Node` objects (a small class *nested* inside `TreeIntDB`, because no other class needs it). `addKey` walks from the root, left for smaller keys

and right for larger, and hangs a new node where the walk falls off the tree; a key that is already present is ignored. `search` walks the same path and counts one comparison per node visited. `TreeIntDB` does **not** override `addSeveral`, so it uses `IntDB`'s loop.

The output confirms each claim. For the linear database, `(bulk copy of 5 keys)` appears: the override ran. For the tree it does not appear: the inherited loop ran, calling `TreeIntDB.addKey` five times. Both databases give the same answers — as they must, since they implement the same abstract idea — but the comparison counts differ, 33 against 22. Let us check 33 by hand, because a number you cannot reproduce is a number you should not trust. The array holds ⟨42,7,19,3,25⟩. `search(19)` looks at 42, 7, 19: three comparisons. `search(8)` looks at all five and fails: five. `findOneOf({8,9,25})` searches 8 (five), 9 (five) and 25 (five, found at the last position): fifteen. `findOneOf({1,2})` searches 1 and 2, five each: ten. The total is $3 + 5 + 15 + 10 = 33$. For the tree, inserting 42, 7, 19, 3, 25 gives the root 42 with left child 7; 7 has left child 3 and right child 19; 19 has right child 25. Every search follows one root-to-leaf path of at most four nodes, which is why the tree needs only 22 comparisons in total. Exercise 7.2 asks you to reproduce the 22.

Listing 7.4 is the Python version.

Listing 7.4 `int_db_demo.py`: the same working database in Python; `LinearIntDB` overrides `add_several` with one call to `list.extend`.

```
"""int_db_demo.py: a working key database, two implementations, one superclass."""
from abc import ABC, abstractmethod


class IntDB(ABC):
    def __init__(self):
        self.comparisons = 0  # counted by search(), so we can compare the two

    # Written once, inherited by every subclass.
    def add_several(self, keys):
        for key in keys:
            self.add_key(key)

    # Written once, inherited by every subclass.
    def find_one_of(self, keys):
        for key in keys:
            if self.search(key):
                return True
        return False

    @abstractmethod
    def add_key(self, key): ...

    @abstractmethod
    def search(self, key): ...

    @abstractmethod
    def size(self): ...

    @abstractmethod
    def name(self): ...


class LinearIntDB(IntDB):
    def __init__(self):
        super().__init__()
        self._table = []

    def add_key(self, key):
        self._table.append(key)

    # Redefinition: one bulk extend instead of one add_key() call per key.
    def add_several(self, keys):
        self._table.extend(keys)
        print(f"  (bulk copy of {len(keys)} keys)")
```

Listing 7.4 (Continued) *int_db_demo.py: the same working database in Python; LinearIntDB overrides add_several with one call to list.extend.*

```
    def search(self, key):
        for value in self._table:
            self.comparisons += 1
            if value == key:
                return True
        return False

    def size(self):
        return len(self._table)

    def name(self):
        return "LinearIntDB (unsorted array)"


class _Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


class TreeIntDB(IntDB):
    def __init__(self):
        super().__init__()
        self._root = None
        self._size = 0

    def add_key(self, key):
        if self._root is None:
            self._root = _Node(key)
            self._size += 1
            return
        cur = self._root
        while True:
            if key == cur.key:
                return  # already present: a set keeps one copy
            elif key < cur.key:
                if cur.left is None:
                    cur.left = _Node(key)
                    self._size += 1
                    return
                cur = cur.left
            else:
                if cur.right is None:
                    cur.right = _Node(key)
                    self._size += 1
```

Listing 7.4 (Continued) *int_db_demo.py: the same working database in Python; LinearIntDB overrides add_several with one call to list.extend.*

```
                    return
                cur = cur.right

    def search(self, key):
        cur = self._root
        while cur is not None:
            self.comparisons += 1
            if key == cur.key:
                return True
            cur = cur.left if key < cur.key else cur.right
        return False

    def size(self):
        return self._size

    def name(self):
        return "TreeIntDB (binary search tree)"


def main():
    for db in (LinearIntDB(), TreeIntDB()):
        print(db.name())
        db.add_several([42, 7, 19, 3, 25])
        print("  size                =", db.size())
        print("  search(19)          =", db.search(19))
        print("  search(8)           =", db.search(8))
        print("  findOneOf({8,9,25}) =", db.find_one_of([8, 9, 25]))
        print("  findOneOf({1,2})    =", db.find_one_of([1, 2]))
        print("  comparisons so far  =", db.comparisons)


if __name__ == "__main__":
    main()
```

```
LinearIntDB (unsorted array)
  (bulk copy of 5 keys)
  size                = 5
  search(19)          = True
  search(8)           = False
  findOneOf({8,9,25}) = True
  findOneOf({1,2})    = False
  comparisons so far  = 33
TreeIntDB (binary search tree)
  size                = 5
  search(19)          = True
  search(8)           = False
  findOneOf({8,9,25}) = True
  findOneOf({1,2})    = False
  comparisons so far  = 22
```

The Python code follows the Java line by line, with three small differences. A Python list grows by itself, so `add_key` is a single `append` and the override of `add_several` a single `extend`. Python has no `@Override` annotation (Python 3.12 added `typing.override`, which only type-checking tools read); a method overrides simply by having the same name. And the node class is a module-level class `_Node`, whose leading underscore says "internal detail" (Section 7.13).

> ⚠ **WARNING.** In Python, a method *overrides* if it has the same **name**; the parameters do not matter. In Java, it overrides only if it has the same name **and the same parameter types**; with different parameter types it is an *overload*, a separate method. The two languages disagree on this point, and it causes real bugs when programmers move between them. Section 7.6 returns to it.

7.4.1 The rules Java checks when you override

Java is strict about overriding, and every rule protects a promise the superclass made to its callers. If a method in class `C` has the same name and parameter types as an inherited method of `B`, then:

1. **The return type must be the same, or a subtype.** If `B.make()` returns `Garment`, then `C.make()` may return `Garment` or `Shirt` (a subclass of `Garment`) but not `String`. The caller who was promised a `Garment` still gets one. A return type narrowed like this is called *covariant*.
2. **The access may not become more restrictive.** A `public` method may not be overridden by a package-private one. Here is what `javac` says to a subclass that tries to override `public void start()` with `void start()`:

```
• Weaker.java:12: error: start() in Derived cannot override start() in Base
    void start() {
         ^
  attempting to assign weaker access privileges; was public
 1 error
```

- The reason: code holding a `Base` variable was promised it could call `start()`; if the object turned out to be a `Derived`, that promise would be broken.

3. **A `final` method cannot be overridden at all.** Writing `final` before a method means "this behavior is fixed for every subclass":

```
• FinalDemo.java:13: error: footer() in FancyReceipt cannot override footer() in
 Receipt
    String footer() {
           ^
  overridden method is final
 1 error
```

- A whole class can be `final` too (`String` is one), and then no class may extend it.

4. **`static` methods and `private` methods are not overridden.** A static method belongs to the class, not to an object, so a subclass method with the same signature merely *hides* it; a private method is invisible to subclasses, so a subclass method with the same name is simply a new, unrelated method. Section 7.7 shows why this matters.
5. **Checked exceptions may not grow.** An overriding method may throw fewer or narrower checked exceptions than the method it overrides, never new or broader ones, for the same reason as rule 2.

All five rules are stated formally in the Java Language Specification [3]. Python enforces none of them; it trusts the programmer, and the price of that trust is that such mistakes surface only when the program runs.

7.5 Constructors and `super`

A subclass object contains a superclass object inside it. A `LinearIntDB` has its own `table` and `size`, but it also has the `comparisons` field declared in `IntDB`, and somebody has to initialize that field. The rule every object-oriented language adopts is simple to state: **the superclass part of an object is built first, by the superclass's own constructor**, and only then does the subclass's constructor run. The subclass may not skip this step, because the superclass knows things about its own fields that the subclass does not.

To see the order, we use a three-level chain from the laundry: every `LaundryItem` gets a numbered tag; a `Garment` is a laundry item of some kind (a towel, a shirt); a `Shirt` is a garment that may or may not be starched. **Listing 7.5** prints a line from every constructor.

Listing 7.5 `ConstructorChain.java`: a three-level chain `LaundryItem`, `Garment`, `Shirt`; each constructor announces itself, and each `describe` extends its parent's text with `super.describe()`.

```
public class ConstructorChain {
    public static void main(String[] args) {
        System.out.println("-- new Shirt(\"blue\", true) --");
        Shirt shirt = new Shirt("blue", true);
        System.out.println(shirt.describe());

        System.out.println("-- new Garment(\"towel\") --");
        Garment towel = new Garment("towel");
        System.out.println(towel.describe());
    }
}

class LaundryItem {
    private static int nextTag = 1;
    protected final int tag;

    LaundryItem() {
        tag = nextTag++;
        System.out.println("  LaundryItem() gives tag #" + tag);
    }

    String describe() {
        return "item #" + tag;
    }
}

class Garment extends LaundryItem {
    protected final String kind;

    Garment(String kind) {
        // The compiler inserts super(); here, so LaundryItem() runs first.
        this.kind = kind;
        System.out.println("  Garment(\"" + kind + "\")");
    }

    @Override
    String describe() {
        return super.describe() + ", a " + kind;
    }
}

class Shirt extends Garment {
    private final boolean starch;
```

Listing 7.5 (Continued) *ConstructorChain.java: a three-level chain LaundryItem, Garment, Shirt; each constructor announces itself, and each describe extends its parent's text with super.describe().*

```
    Shirt(String color, boolean starch) {
        super(color + " shirt");  // must be the first statement
        this.starch = starch;
        System.out.println("  Shirt(..., starch=" + starch + ")");
    }

    @Override
    String describe() {
        return super.describe() + (starch ? ", starched" : ", no starch");
    }
}
```

```
-- new Shirt("blue", true) --
  LaundryItem() gives tag #1
  Garment("blue shirt")
  Shirt(..., starch=true)
item #1, a blue shirt, starched
-- new Garment("towel") --
  LaundryItem() gives tag #2
  Garment("towel")
item #2, a towel
```

Read the first block of output from the top. We asked for a `Shirt`, yet the first line printed comes from `LaundryItem`, then `Garment`, and only then `Shirt`. **Figure 7.2** shows why. Calling `new Shirt("blue", true)` starts the `Shirt` constructor; its very first statement is `super(color + " shirt")`, which calls the `Garment` constructor with the argument `"blue shirt"`. The `Garment` constructor begins with a call we cannot see: when a constructor does not start with `super(...)` or `this(...)`, the Java compiler inserts `super();` — a call to the superclass's no-argument constructor — on its behalf. So `LaundryItem()` runs, and it, too, begins with an invisible `super()`, which reaches the constructor of `Object`, the root of every Java class (Section 7.8). Then the bodies complete in reverse order: `LaundryItem` assigns tag 1 and prints; `Garment` stores its kind and prints; `Shirt` stores the starch flag and prints.

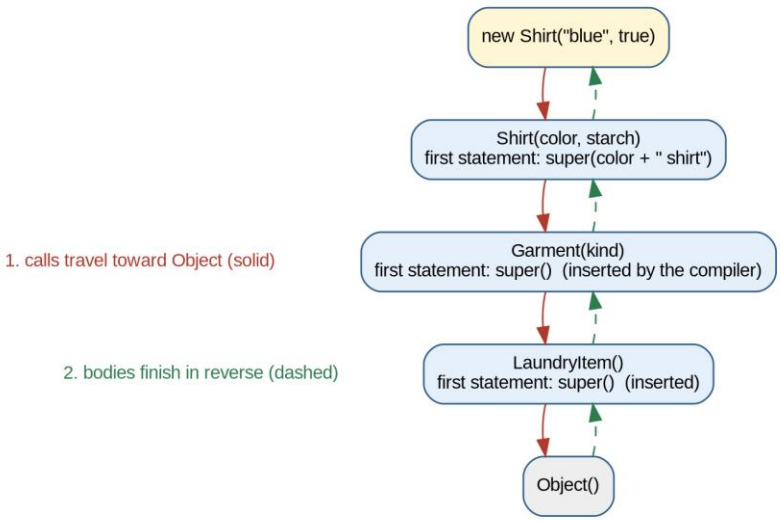


Figure 7.2 Constructor chaining for `new Shirt("blue", true)`. The calls travel up the hierarchy to `Object` first (solid arrows); the constructor bodies then finish in reverse order (dashed arrows), so the most general part of the object is always ready before the most specific part.

Three rules fall out of this, and Java enforces all of them:

1. `super(...)` **must be the first statement of a constructor**, if it appears at all. Nothing may happen to a `Shirt` before its `Garment` part exists. (Since Java 25 the rule is relaxed slightly by *flexible constructor bodies* [4]: statements that do not use the object under construction may come before the call; the idea is unchanged.)
2. **If you do not write `super(...)`, the compiler writes `super()` for you** — and that call must be legal. `Garment` has *no* no-argument constructor; its only constructor needs a `String`. So if `Shirt`'s constructor forgot its `super(color + " shirt")` line, the invisible `super()` would have nothing to call. Delete that line from **Listing 7.5** and `javac` says:

```
• ConstructorChain.java:45: error: constructor Garment in class Garment cannot be
  applied to given types;
      Shirt(String color, boolean starch) {
                                          ^
    required: String
    found:    no arguments
    reason: actual and formal argument lists differ in length
  1 error
```

   - The message points at `Shirt`'s constructor and complains about `Garment`'s: exactly the invisible call.
3. **Constructors are not inherited.** `Shirt` does not get a `Shirt(String kind)` constructor just because `Garment` has one. Each class states how its own objects are built.

The keyword `super` has a second use, visible in the three `describe` methods. Inside an overriding method, `super.describe()` calls the *superclass's* version of the same method — the one being overridden. `Shirt.describe()` asks `Garment.describe()` for its text, which in turn asks `LaundryItem.describe()`, and each level appends its own part: `item #1`, then `, a blue shirt`, then `, starched`. This pattern, "do what my parent does, then a little more", is the most common reason to call `super` from a method, and we will meet it again in the Animal program and the laundry receipt.

Listing 7.6 is the Python version.

Listing 7.6 `constructor_chain.py`: the same chain in Python, plus a class that forgets to call `super().__init__()`.

```
"""constructor_chain.py: who runs first when a Shirt is built?"""


class LaundryItem:
    _next_tag = 1

    def __init__(self):
        self.tag = LaundryItem._next_tag
        LaundryItem._next_tag += 1
        print(f"  LaundryItem() gives tag #{self.tag}")

    def describe(self):
        return f"item #{self.tag}"


class Garment(LaundryItem):
    def __init__(self, kind):
        super().__init__()  # Python never inserts this for you
        self.kind = kind
        print(f'  Garment("{kind}")')

    def describe(self):
        return super().describe() + f", a {self.kind}"


class Shirt(Garment):
    def __init__(self, color, starch):
        super().__init__(color + " shirt")
        self.starch = starch
        print(f"  Shirt(..., starch={str(starch).lower()})")

    def describe(self):
        return super().describe() + (", starched" if self.starch else ", no starch")


class ForgetfulGarment(LaundryItem):
    def __init__(self, kind):
        self.kind = kind  # forgot super().__init__(): no tag is ever set


def main():
    print('-- new Shirt("blue", true) --')
    shirt = Shirt("blue", True)
    print(shirt.describe())
```

Listing 7.6 (Continued) *constructor_chain.py: the same chain in Python, plus a class that forgets to call super().__init__().*

```
    print('-- new Garment("towel") --')
    towel = Garment("towel")
    print(towel.describe())

    print("-- ForgetfulGarment(\"sock\") --")
    sock = ForgetfulGarment("sock")
    try:
        print(sock.describe())
    except AttributeError as e:
        print("  AttributeError:", e)


if __name__ == "__main__":
    main()
```

```
-- new Shirt("blue", true) --
  LaundryItem() gives tag #1
  Garment("blue shirt")
  Shirt(..., starch=true)
item #1, a blue shirt, starched
-- new Garment("towel") --
  LaundryItem() gives tag #2
  Garment("towel")
item #2, a towel
-- ForgetfulGarment("sock") --
  AttributeError: 'ForgetfulGarment' object has no attribute 'tag'
```

The first two blocks are identical to Java's, and the code maps one to one: `super().__init__(color + " shirt")` is Python's `super(color + " shirt")`. But one rule is different, and the third block shows the consequence. **Python never inserts the call for you.** If a subclass defines `__init__` and does not call `super().__init__(...)`, the superclass's `__init__` simply never runs. `ForgetfulGarment` makes that mistake: its object gets a `kind` but never a `tag`, and the program only discovers the problem later, when `describe()` tries to read `self.tag` and raises `AttributeError`. The error appears far from the cause, which is what makes this one of the most common Python inheritance bugs. The habit to form is easy: **the first line of every subclass `__init__` is `super().__init__(...)`.**

(Conversely, if a Python subclass does not define `__init__` at all, it inherits the superclass's `__init__` like any other method. `Cat()` in the next section works that way.)

7.6 Polymorphism: One Call, Many Behaviors

The word *polymorphism* comes from Greek and means "many forms". In object-oriented programming it describes the situation we have now seen several times: many classes, related by inheritance, each giving its own form to the same method, so that one call in the program can take many forms at run time. Inheritance lets a class *receive* attributes and methods from another; polymorphism lets it *use* those methods to perform the same action in its own way.

The classic example is a superclass `Animal` with a method `sound()`, and subclasses `Cat`, `Cow` and `Dog` that each make their own sound. **Figure 7.3** shows the design, and **Listing 7.7** the complete program. It is worth studying slowly, because it places the two features that beginners confuse most — *overloading* and *overriding* — side by side in one class.

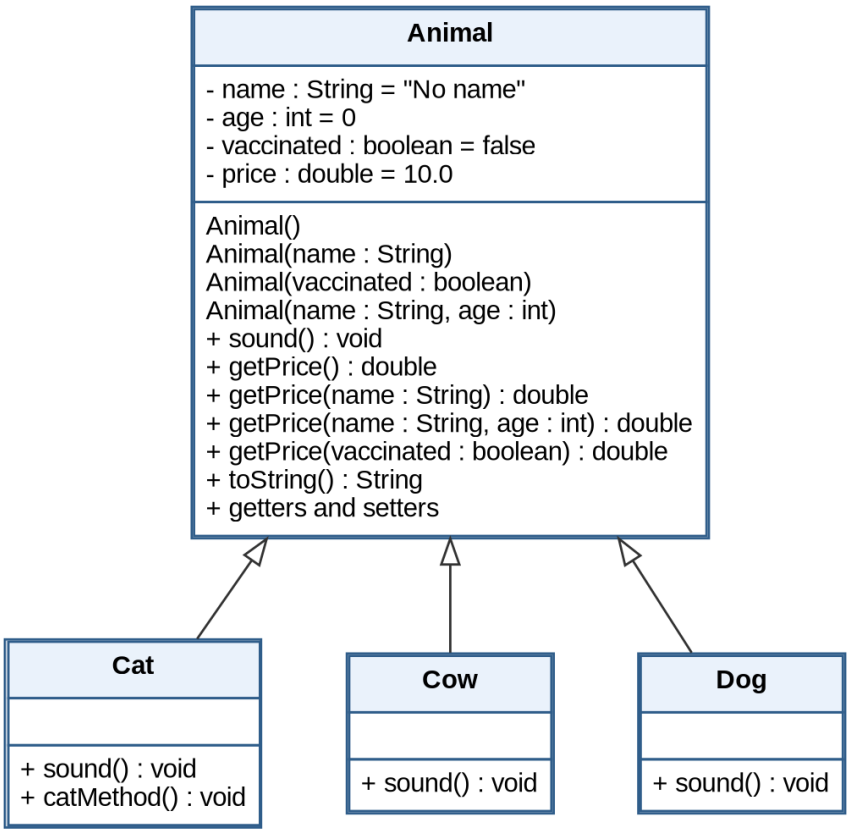


Figure 7.3 UML class diagram of the Animal program. `Animal` has four overloaded constructors and four overloaded `getPrice` methods; `Cat`, `Cow` and `Dog` each override `sound()`, and `Cat` adds a method of its own. A hollow triangle points from a subclass to its superclass; `-` marks a private member and `+` a public one.

Listing 7.7 `animal/AnimalTest.java`: `Animal` with overloaded constructors and `getPrice` methods, three subclasses that override `sound()`, and a test program that exercises both overloading and overriding.

```
package animal;

public class AnimalTest {
    public static void main(String[] args) {
        Animal animal = new Animal();  // Create an Animal object
        // Overloading
        Animal animal2 = new Animal("Bibi");  // With a name
        Animal animal3 = new Animal(true);    // With a vaccinated status
        Animal animal4 = new Animal("Chiki", 3);  // With a name & age
        System.out.println();

        Animal cat = new Cat();  // Create a Cat object
        Animal cow = new Cow();  // Create a Cow object
        Animal dog = new Dog();  // Create a Dog object
        System.out.println();

        // Polymorphism
        animal.sound();
        cat.sound();
        cow.sound();
        dog.sound();
        System.out.println();

        // Overloading
        System.out.println(animal);
        System.out.println("Age: " + animal.getAge());
        System.out.println("Price: " + animal.getPrice());
        System.out.println("Vaccinated: " + animal.isVaccinated());
        System.out.println();

        System.out.println(animal2);
        System.out.println("Age: " + animal2.getAge());
        System.out.println("Price: " + animal2.getPrice(animal2.getName()));
        System.out.println("Vaccinated: " + animal2.isVaccinated());
        System.out.println();

        System.out.println(animal3);
        System.out.println("Age: " + animal3.getAge());
        System.out.println("Price: " + animal3.getPrice(animal3.isVaccinated()));
        System.out.println("Vaccinated: " + animal3.isVaccinated());
        System.out.println();
```

Listing 7.7 (Continued) *animal/AnimalTest.java: Animal with overloaded constructors and getPrice methods, three subclasses that override sound(), and a test program that exercises both overloading and overriding.*

```
        System.out.println(animal4);
        System.out.println("Age: " + animal4.getAge());
        System.out.println("Price: " + animal4.getPrice(animal4.getName(),
                animal4.getAge()));
        System.out.println("Vaccinated: " + animal4.isVaccinated());
        System.out.println();

        System.out.print("Showing that a superclass can call ");
        System.out.println("a subclass method");
        System.out.println("---------------------------------------------------");
        System.out.println("a1 is defined by superclass: Animal, instantiated by
superclass: Animal");
        Animal a1 = new Animal();
        System.out.println("b1 is defined by subclass: Cat, instantiated by subclass:
Cat");
        Cat b1 = new Cat();
        System.out.println("b1 calls the Cat's method: catMethod()");
        b1.catMethod();
        // System.out.println("Try: a1 calls the subclass' method: catMethod() by casting
to Cat");
        // ((Cat) a1).catMethod(); // Exception will happen
        System.out.println("c1 is defined by superclass: Animal, instantiated by subclass:
Cat");
        Animal c1 = new Cat();
        // c1.catMethod(); // The subclass' method: catMethod() is undefined for the type
Animal
        ((Cat) c1).catMethod();  // Need to cast, so that an instance of superclass can
call
                                 // the subclass' method
    }
}

class Animal {
    // Information hiding = encapsulation
    // This class property set to be private -> cannot be accessed
    // directly. Only can be accessed by getter(s) & setter(s)
    private String name = "No name";  // Default name: No name
    private int age = 0;  // Default age: 0
    private boolean vaccinated = false;  // Default: false
    private double price = 10.0;  // Default: 10.0

    // Overloading: a class has more than one method of
    // the same name & their parameter(s) are different

    // Constructor: special case of function, where its name
    // is the same with the class' name
    Animal() {  // Overloading
        System.out.println("Animal: Default constructor");
    }
```

Listing 7.7 (Continued) *animal/AnimalTest.java: Animal with overloaded constructors and getPrice methods, three subclasses that override sound(), and a test program that exercises both overloading and overriding.*

```
    Animal(String name) {  // Overloading
        this.name = name;
        System.out.println("Constructor with name");
    }

    Animal(boolean vaccinated) {  // Overloading
        this.vaccinated = vaccinated;
        System.out.println("Constructor with vaccinated status");
    }

    Animal(String name, int age) {  // Overloading
        this.name = name;
        this.age = age;
        System.out.println("Constructor with name & age");
    }

    // Overriding: same method name & same parameter(s)
    // By using overriding, Run Time Polymorphism can be achieved
    public void sound() {  // Polymorphism
        System.out.println("Animal makes a sound");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isVaccinated() {
        return vaccinated;
    }

    public void setVaccinated(boolean vaccinated) {
        this.vaccinated = vaccinated;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
```

Listing 7.7 (Continued) *animal/AnimalTest.java: Animal with overloaded constructors and getPrice methods, three subclasses that override sound(), and a test program that exercises both overloading and overriding.*

```
    public double getPrice() {  // Overloading
        return price;
    }

    public double getPrice(String name) {  // Overloading
        if (name.equals("No name")) {
            return price + 100;
        } else {
            return price + 110;
        }
    }

    public double getPrice(String name, int age) {  // Overloading
        if (name.equals("No name")) {
            if (age > 1) {
                return price + 200;
            } else {
                return price + 150;
            }
        } else {
            if (age > 1) {
                return price + 300;
            } else {
                return price + 200;
            }
        }
    }

    public double getPrice(boolean vaccinated) {  // Overloading
        if (vaccinated) {
            return price + 400;
        } else {
            return price;
        }
    }

    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public String toString() {  // Used for printing an instance
        return "[" + name + "]";
    }
}
```

Listing 7.7 (Continued) *animal/AnimalTest.java: Animal with overloaded constructors and getPrice methods, three subclasses that override sound(), and a test program that exercises both overloading and overriding.*

```
class Cat extends Animal {
    // Overriding: same method name & same parameter(s)
    // By using overriding, Run Time Polymorphism can be achieved
    @Override
    public void sound() {  // Polymorphism
        System.out.println("Cat says: meow");
    }

    // An example of Cat's method/function
    public void catMethod() {
        System.out.println("Cat: catMethod");
    }
}

class Cow extends Animal {
    @Override
    public void sound() {  // Polymorphism
        System.out.println("Cow says: moo");
    }
}

class Dog extends Animal {
    @Override
    public void sound() {  // Polymorphism
        System.out.println("Dog says: woof");
    }
}
```

Compile and run from the `java` folder with `javac -d out animal/AnimalTest.java` and `java -cp out animal.AnimalTest`:

```
Animal: Default constructor
Constructor with name
Constructor with vaccinated status
Constructor with name & age

Animal: Default constructor
Animal: Default constructor
Animal: Default constructor

Animal makes a sound
Cat says: meow
Cow says: moo
Dog says: woof

[No name]
Age: 0
Price: 10.0
Vaccinated: false

[Bibi]
Age: 0
Price: 120.0
Vaccinated: false

[No name]
Age: 0
Price: 410.0
Vaccinated: true

[Chiki]
Age: 3
Price: 310.0
Vaccinated: false

Showing that a superclass can call a subclass method
--------------------------------------------------
a1 is defined by superclass: Animal, instantiated by superclass: Animal
Animal: Default constructor
b1 is defined by subclass: Cat, instantiated by subclass: Cat
Animal: Default constructor
b1 calls the Cat's method: catMethod()
Cat: catMethod
c1 is defined by superclass: Animal, instantiated by subclass: Cat
Animal: Default constructor
Cat: catMethod
```

7.6.1 Reading the Animal class

- **Encapsulated fields.** The four fields are `private`: no code outside `Animal` may touch them. Each has a default value — `"No name"`, `0`, `false`, `10.0` — that holds unless a constructor changes it. The outside world reads and changes them only through the public *getter* and *setter* methods further down (`getName`, `setName`, and so on). Hiding data behind methods in this way is called *encapsulation*, or *information hiding*, and Section 7.14 explains why it matters.
- **Overloaded constructors.** `Animal` has four constructors: with no arguments, with a `String`, with a `boolean`, and with a `String` and an `int`. They share a name and differ in their parameter lists: *overloading*, as in Chapter 6. The compiler picks the constructor whose parameters match the arguments, so `new Animal("Bibi")` calls `Animal(String name)` and `new Animal(true)` calls `Animal(boolean vaccinated)`. The first four lines of output are the four constructors announcing themselves.
- **Overloaded `getPrice`.** The same trick gives four `getPrice` methods. The base price is 10.0; the version that takes a name adds 100 for an unnamed animal and 110 for a named one; the

version that takes a name and an age adds between 150 and 300; the version that takes the vaccination flag adds 400 if the animal is vaccinated. Check the output: Bibi, named, costs 10 + 110 = 120; the vaccinated animal costs 10 + 400 = 410; Chiki, named and three years old, costs 10 + 300 = 310.

- **`sound()` is meant to be overridden.** `Animal.sound()` prints a generic line. Each subclass replaces it. That is *overriding*.
- **`toString()`** is also an override — of a method every Java class inherits from `Object` (Section 7.8). `System.out.println(animal)` calls it automatically, which is why the output shows `[No name]` and `[Bibi]`.

The three subclasses are tiny. `Cat`, `Cow` and `Dog` each declare `extends Animal` and override `sound()`. None declares a constructor, so each gets a *default constructor* from the compiler — one that takes no arguments and simply calls `super()`. That is why creating the cat, the cow and the dog prints `Animal: Default constructor` three times: the only constructor that prints anything is `Animal()`, reached through the invisible `super()`. `Cat` also adds `catMethod()`, a method that `Animal` does not have.

7.6.2 Reading the test program

The block commented `// Polymorphism` is the point of the program. The four variables `animal`, `cat`, `cow` and `dog` are all declared with type `Animal`. The four calls look identical — `animal.sound()`, `cat.sound()`, and so on — and yet they print four different lines. The compiler did not decide which `sound()` to run; it could not, because it only knows each variable is an `Animal`. The decision was made at run time by looking at the actual object. That is **run-time polymorphism**, and the next section takes it apart.

The final block shows the other side of the coin. `a1` is declared `Animal` and holds an `Animal`; `b1` is declared `Cat` and holds a `Cat`, so `b1.catMethod()` is fine. `c1` is declared `Animal` but holds a `Cat`. Can we write `c1.catMethod()`? The object certainly has that method. But remove the `//` in front of that line and the compiler refuses:

```
animal/AnimalTest.java:63: error: cannot find symbol
        c1.catMethod(); // The subclass' method: catMethod() is undefined for the type
Animal
          ^
  symbol:   method catMethod()
  location: variable c1 of type Animal
1 error
```

The compiler reasons from the declared type, `Animal`, and `Animal` has no `catMethod`. To call it we must *cast*: `((Cat) c1).catMethod()` tells the compiler "trust me, this `Animal` is really a `Cat`". The cast is checked when the program runs. If we lied — as in the commented-out line `((Cat) a1).catMethod()`, where `a1` is a plain `Animal` — the program stops with a `ClassCastException`. Section 7.7 shows a safer way to write such casts.

7.6.3 Overloading versus overriding

The two words sound alike and both involve "several methods with one name". **Table 7.1** sets them side by side.

Table 7.1 Overloading and overriding compared.

	Overloading	Overriding
Where	In one class (or a class and its subclass)	In a subclass, replacing a superclass method
Same name?	Yes	Yes
Same parameter types?	**No** — they must differ	**Yes** — they must match
Who chooses the method	The compiler, from the declared types of the arguments	The running program, from the class of the actual object
When it is chosen	Compile time (*static* binding)	Run time (*dynamic* binding)
Example in **Listing 7.7**	Four `Animal(...)` constructors, four `getPrice(...)`	`sound()` in `Cat`, `Cow`, `Dog`; `toString()` in `Animal`
Python	Not supported: a second `def` with the same name replaces the first; use default or keyword arguments	Supported: same name in the subclass

The last row deserves emphasis: Python has **no overloading**. A second `def get_price` in the same class simply replaces the first. Python programmers get the same flexibility from default and keyword arguments, as **Listing 7.8** shows.

Listing 7.8 `animal_test.py`: the Animal program in Python; one `__init__` and one `get_price` with optional keyword parameters replace Java's four overloads each.

```
"""animal_test.py: Animal, Cat, Cow and Dog; overriding, and Python's answer to
overloading."""


class Animal:
    # One __init__ with optional parameters replaces Java's four constructors.
    def __init__(self, name=None, age=None, vaccinated=None):
        self._name = "No name"   # Default name: No name
        self._age = 0            # Default age: 0
        self._vaccinated = False  # Default: False
        self._price = 10.0       # Default: 10.0
        if name is not None and age is not None:
            self._name, self._age = name, age
            print("Constructor with name & age")
        elif name is not None:
            self._name = name
            print("Constructor with name")
        elif vaccinated is not None:
            self._vaccinated = vaccinated
            print("Constructor with vaccinated status")
        else:
            print("Animal: Default constructor")

    # Overriding: same method name & same parameter(s)
    def sound(self):
        print("Animal makes a sound")

    @property
    def name(self):
        return self._name

    @property
    def age(self):
        return self._age

    @property
    def vaccinated(self):
        return self._vaccinated
```

Listing 7.8 (Continued) *animal_test.py: the Animal program in Python; one __init__ and one get_price with optional keyword parameters replace Java's four overloads each.*

```
    # One method with keyword parameters replaces Java's four getPrice overloads.
    def get_price(self, name=None, age=None, vaccinated=None):
        if name is not None and age is not None:
            if name == "No name":
                return self._price + (200 if age > 1 else 150)
            return self._price + (300 if age > 1 else 200)
        if name is not None:
            return self._price + (100 if name == "No name" else 110)
        if vaccinated is not None:
            return self._price + 400 if vaccinated else self._price
        return self._price

    def __str__(self):  # Used for printing an instance
        return "[" + self._name + "]"


class Cat(Animal):
    def sound(self):  # Polymorphism
        print("Cat says: meow")

    def cat_method(self):  # An example of Cat's own method
        print("Cat: catMethod")


class Cow(Animal):
    def sound(self):
        print("Cow says: moo")


class Dog(Animal):
    def sound(self):
        print("Dog says: woof")


def show(animal, price):
    print(animal)
    print("Age:", animal.age)
    print("Price:", price)
    print("Vaccinated:", str(animal.vaccinated).lower())
    print()
```

Listing 7.8 (Continued) *animal_test.py: the Animal program in Python; one __init__ and one get_price with optional keyword parameters replace Java's four overloads each.*

```
def main():
    animal = Animal()
    animal2 = Animal(name="Bibi")
    animal3 = Animal(vaccinated=True)
    animal4 = Animal(name="Chiki", age=3)
    print()

    cat, cow, dog = Cat(), Cow(), Dog()
    print()

    for a in (animal, cat, cow, dog):  # Polymorphism
        a.sound()
    print()

    show(animal, animal.get_price())
    show(animal2, animal2.get_price(name=animal2.name))
    show(animal3, animal3.get_price(vaccinated=animal3.vaccinated))
    show(animal4, animal4.get_price(name=animal4.name, age=animal4.age))

    print("A variable has no declared type: only the object decides")
    print("-----------------------------------------------------")
    a1 = Animal()
    c1 = Cat()
    c1.cat_method()  # no cast needed: the object is a Cat
    try:
        a1.cat_method()  # the object is a plain Animal
    except AttributeError as e:
        print("AttributeError:", e)


if __name__ == "__main__":
    main()
```

```
Animal: Default constructor
Constructor with name
Constructor with vaccinated status
Constructor with name & age

Animal: Default constructor
Animal: Default constructor
Animal: Default constructor

Animal makes a sound
Cat says: meow
Cow says: moo
Dog says: woof

[No name]
Age: 0
Price: 10.0
Vaccinated: false

[Bibi]
Age: 0
Price: 120.0
Vaccinated: false

[No name]
Age: 0
Price: 410.0
Vaccinated: true

[Chiki]
Age: 3
Price: 310.0
Vaccinated: false

A variable has no declared type: only the object decides
--------------------------------------------------
Animal: Default constructor
Animal: Default constructor
Cat: catMethod
AttributeError: 'Animal' object has no attribute 'cat_method'
```

Everything up to the last block matches Java's output exactly. The single `__init__` takes three optional parameters, each defaulting to `None` ("not given"), and decides which Java constructor it imitates from the arguments that arrived. The call sites describe themselves — `Animal(vaccinated=True)` says more than Java's `new Animal(true)`. `get_price` works the same way. The getters became *properties* (`@property`), which let the caller write `animal.age` instead of `animal.get_age()`; Section 7.14 explains them.

The last block is where the languages part company. A Python variable has no declared type, so there is nothing to cast: `c1.cat_method()` simply works, because the object is a `Cat`. And `a1.cat_method()`, which in Java would not even compile, runs in Python and fails with `AttributeError` at the moment of the call. Python's motto for this style is "if it walks like a duck and quacks like a duck, it is a duck" — *duck typing*: the only question is whether the object has the method, never what the variable was declared to be.

7.7 How Dynamic Dispatch Really Works: a Picture of Memory

We have said several times that "the actual object decides which method runs". That sentence is true but vague, and vagueness is where misconceptions breed. In this section we make it precise,

first in Java and then in Python, by drawing what is in the computer's memory while the program runs.

7.7.1 Two types for every variable

In Java every expression has two types, and it is essential to keep them apart:

> **DEFINITION.** The **static type** (or *declared type*) of a variable is the type written in its declaration; it is fixed when the program is compiled. The **dynamic type** (or *run-time type*) is the class of the object the variable refers to at a given moment while the program runs. The dynamic type is always the static type or one of its subclasses.

After `Machine m = new Washer("W-01");`, the static type of `m` is `Machine` and its dynamic type is `Washer`. Assigning a subclass object to a superclass variable, as here, is called **upcasting**; it is always safe and happens automatically, because a `Washer` is a `Machine`. Going the other way — treating a `Machine` variable as a `Washer` — is **downcasting**; it is not automatic, and not always safe.

The compiler and the running program use the two types for different jobs:

- The **compiler uses the static type** to decide whether a call is *allowed*. `m.start()` compiles because `Machine` has a `start()`. `m.setHeat(3)` would not compile, even if `m` held a `Dryer`, because `Machine` has no `setHeat`.
- The **running program uses the dynamic type** to decide *which* overriding method to run. `m.start()` runs `Washer.start()` because the object is a `Washer`.

This division of labor has a name: **dynamic dispatch** (also *late binding*). It applies to *instance methods that can be overridden*. It does **not** apply to fields, to `static` methods or to `private` methods, all of which are chosen by the compiler from the static type. **Listing 7.9** shows each case with a small laundry machine hierarchy.

Listing 7.9 `DispatchDemo.java`: dynamic dispatch of `start()`, the non-dispatch of a hidden field and a hidden static method, a safe downcast with `instanceof`, and an unsafe one.

```
public class DispatchDemo {
    public static void main(String[] args) {
        Machine m = new Washer("W-01");  // static type Machine, dynamic type Washer
        m.start();                       // chosen at run time: Washer.start()
        System.out.println("m.label  = " + m.label);   // fields: chosen at compile time
        System.out.println("m.kind() = " + m.kind());  // static: chosen at compile time
        System.out.println("m.describe() = " + m.describe());
        System.out.println();

        Machine[] floor = {new Washer("W-02"), new Dryer("D-01"), new Machine("M-00")};
        for (Machine x : floor) {
            x.start();  // one line of code, three different behaviors
        }
        System.out.println();

        for (Machine x : floor) {
            if (x instanceof Dryer d) {  // check first, then use as a Dryer
                d.setHeat(3);
            }
        }
        try {
            Dryer wrong = (Dryer) floor[0];  // floor[0] is a Washer
            wrong.setHeat(1);
        } catch (ClassCastException e) {
            System.out.println("Casting a Washer to Dryer: " +
e.getClass().getSimpleName());
        }
        System.out.println("floor[1] is really a " + floor[1].getClass().getSimpleName());
    }
}

class Machine {
    protected final String id;
    String label = "machine";

    Machine(String id) {
        this.id = id;
    }

    void start() {
        System.out.println(id + ": generic machine starts");
    }

    static String kind() {
        return "Machine.kind()";
    }
```

Listing 7.9 (Continued) *DispatchDemo.java: dynamic dispatch of start(), the non-dispatch of a hidden field and a hidden static method, a safe downcast with instanceof, and an unsafe one.*

```
    String describe() {
        return id + " is a " + label;  // reads Machine's label field
    }
}

class Washer extends Machine {
    String label = "washer";  // hides Machine.label; does NOT override it

    Washer(String id) {
        super(id);
    }

    @Override
    void start() {
        System.out.println(id + ": fill water, spin at 1200 rpm");
    }

    static String kind() {  // hides Machine.kind(); static methods do not override
        return "Washer.kind()";
    }
}

class Dryer extends Machine {
    private int heat = 2;

    Dryer(String id) {
        super(id);
    }

    @Override
    void start() {
        System.out.println(id + ": tumble with heat level " + heat);
    }

    void setHeat(int heat) {
        this.heat = heat;
        System.out.println(id + ": heat set to " + heat);
    }
}
```

```
W-01: fill water, spin at 1200 rpm
m.label  = machine
m.kind() = Machine.kind()
m.describe() = W-01 is a machine

W-02: fill water, spin at 1200 rpm
D-01: tumble with heat level 2
M-00: generic machine starts

D-01: heat set to 3
Casting a Washer to Dryer: ClassCastException
floor[1] is really a Dryer
```

Work through the first block, which contains three surprises for the unwary.

1. `m.start()` printed the *washer's* line: dynamic dispatch, as expected.
2. `m.label` printed `machine`, not `washer`. `Washer` declares its own field called `label`, but a field with the same name as a superclass field does not *override* it; it *hides* it. The object actually contains **two** `label` fields, one from each class, and the compiler picks one by the static type of the expression, here `Machine`. Even inside `describe()` — a `Machine` method — `label` means

`Machine`'s field, which is why `describe()` said `W-01 is a machine`. Hidden fields are almost always a mistake; the cure is never to redeclare a superclass field in a subclass.

3. `m.kind()` printed `Machine.kind()`. `kind` is `static`: it belongs to the class, not to any object, so there is no object to dispatch on, and the compiler binds the call to the static type. (Calling a static method through a variable is legal but misleading; `javac -Xlint` warns "static method should be qualified by type name", and so should you.)

The second block is polymorphism doing its daily work. The array `floor` has static element type `Machine`, but holds a `Washer`, a `Dryer` and a plain `Machine`. The loop body is one line, `x.start()`, and it produced three behaviors. Adding a fourth kind of machine next year will not require touching this loop.

The third block shows downcasting. We want to set the heat on every dryer, but `setHeat` exists only in `Dryer`. The test `x instanceof Dryer d` asks "is the dynamic type of `x` a `Dryer` (or a subclass of it)?", and if so it declares a new variable `d` of static type `Dryer` that refers to the same object. Inside the `if`, `d.setHeat(3)` compiles and is safe. This combined test-and-cast form, called *pattern matching for `instanceof`*, has been standard since Java 16. The `try` block does what you should never do: it casts `floor[0]` to `Dryer` without checking. The object is a `Washer`, the cast fails at run time, and Java throws `ClassCastException`. Finally, `getClass()` — a method every object inherits from `Object` — reports the dynamic type.

Table 7.2 summarizes which member is chosen by which type.

Table 7.2 What is decided when: only overridable instance methods are dispatched on the dynamic type.

Member	Chosen by	At	In
Instance method, overridable	Dynamic type of the object	Run time	`m.start()` runs `Washer.start`
`private` method	Declaring class	Compile time	—
`static` method	Static type of the expression	Compile time	`m.kind()` runs `Machine.kind`
Field (instance or static)	Static type of the expression	Compile time	`m.label` reads `Machine.label`
Overload among several methods	Static types of the arguments	Compile time	the four `getPrice` of **Listing 7.7**

7.7.2 The memory picture: method tables

How does the running program find `Washer.start` so quickly? It cannot afford to search through class definitions on every call; a price loop may call `price()` millions of times. **Figure 7.4** shows the idea used by the Java virtual machine and by C++ compilers.

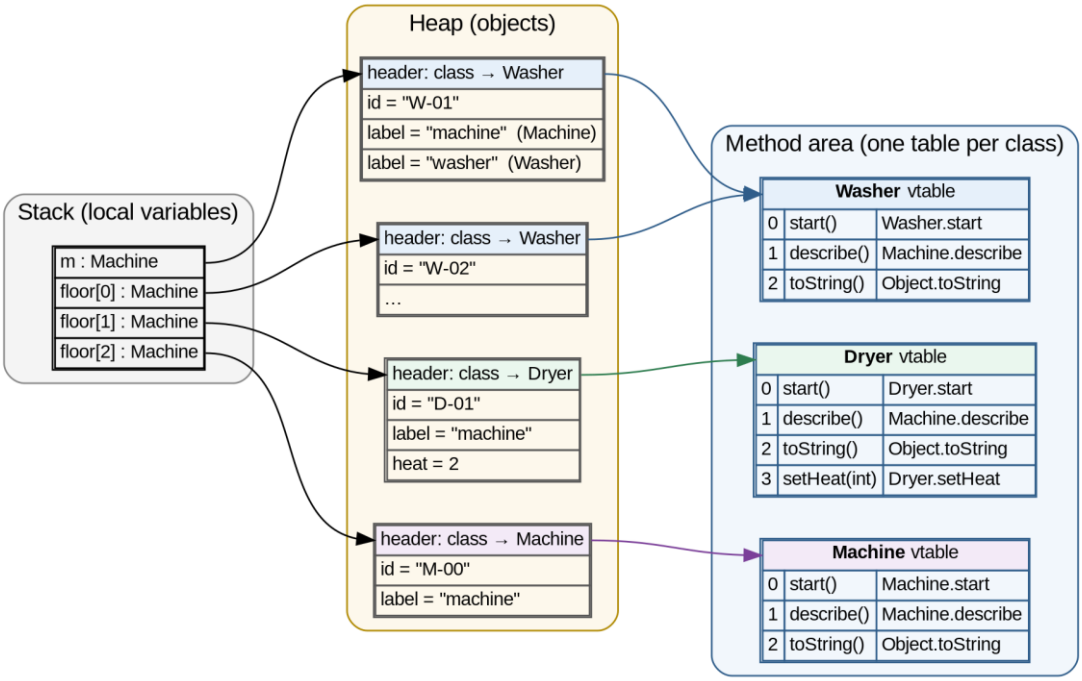


Figure 7.4 Memory while **Listing 7.9** runs. Local variables on the stack hold references to objects on the heap; every object's header points to its class; every class has one method table (*vtable*) whose slot k holds the code for the k-th overridable method. Subclasses copy their parent's slots and overwrite the ones they override.

Read it from left to right.

- **The stack** holds the local variables of `main`: `m` and the elements of `floor`. Each is a *reference*, an arrow to an object; all are declared `Machine`.
- **The heap** holds the objects. Every object begins with a small *header* that contains, among other things, a pointer to the object's class. After the header come the fields — all of them, including those declared in superclasses. Look at the first `Washer`: it carries `id`, the `label` declared in `Machine`, *and* the `label` declared in `Washer`. That is the hidden-field surprise made visible.
- **The method area** holds one table per class, called a **virtual method table** or **vtable**. Each overridable method is given a fixed *slot number* when the classes are loaded: in this hierarchy `start()` is slot 0, `describe()` slot 1, `toString()` slot 2. A subclass's table starts as a copy of its superclass's table; every method the subclass overrides replaces the entry in its slot, and every new method is appended at the end. So `Washer`'s slot 0 holds `Washer.start`, while its slot 1 still holds the inherited `Machine.describe`. `Dryer` overrides slot 0 and appends `setHeat` as slot 3.

Now a call `x.start()` is compiled into three machine-level steps, and none of them is a search:

$$\text{code to run} = \text{vtable}(\text{class}(x))[\,\text{slot}(\text{start})\,] \tag{7.1}$$

Follow the reference `x` to the object; follow the object's header to its class's vtable; jump to the code in the slot that the compiler fixed in advance. Equation (7.1) costs the same few memory reads whether the hierarchy has two classes or two hundred. This is why dynamic dispatch is cheap enough to use everywhere. (The Java Virtual Machine Specification describes the required *behavior* of the `invokevirtual` instruction, not this particular data structure [5]; real JVMs use vtables, and go further by noticing when a call site only ever meets one class and calling that method directly.)

The picture also explains the rules of **Table 7.2**. A field is read at a fixed offset inside the object chosen by the compiler, so the compiler's static view decides which `label` is read. A static or private method never enters a vtable, so it is called directly. And the compiler's check that `m.setHeat(3)` is illegal is simply the observation that `Machine`'s table has no slot called `setHeat`.

7.7.3 The same idea in Python

Python has no static types to worry about, and its lookup is organized differently, but the outcome is the same. **Listing 7.10** repeats the experiment.

Listing 7.10 `dispatch_demo.py`: dynamic lookup in Python, with a helper that walks the method resolution order exactly as Python does.

```
"""dispatch_demo.py: how Python finds the method to run."""


class Machine:
    label = "machine"  # a class attribute

    def __init__(self, machine_id):
        self.id = machine_id

    def start(self):
        print(f"{self.id}: generic machine starts")

    def describe(self):
        return f"{self.id} is a {self.label}"  # self.label is looked up dynamically


class Washer(Machine):
    label = "washer"

    def start(self):
        print(f"{self.id}: fill water, spin at 1200 rpm")


class Dryer(Machine):
    def __init__(self, machine_id):
        super().__init__(machine_id)
        self.heat = 2

    def start(self):
        print(f"{self.id}: tumble with heat level {self.heat}")

    def set_heat(self, heat):
        self.heat = heat
        print(f"{self.id}: heat set to {heat}")


def where_is(cls, name):
    """Walk the MRO exactly as Python does and report the class that defines name."""
    for klass in cls.__mro__:
        if name in klass.__dict__:
            return klass.__name__
    return None
```

Listing 7.10 (Continued) *dispatch_demo.py: dynamic lookup in Python, with a helper that walks the method resolution order exactly as Python does.*

```
def main():
    m: Machine = Washer("W-01")  # the annotation is a hint, not a rule
    m.start()
    print("m.label =", m.label)  # attributes are dynamic too, unlike Java fields
    print("m.describe() =", m.describe())
    print()

    floor = [Washer("W-02"), Dryer("D-01"), Machine("M-00")]
    for x in floor:
        x.start()  # one line of code, three different behaviors
    print()

    for x in floor:
        if isinstance(x, Dryer):
            x.set_heat(3)
    try:
        floor[0].set_heat(1)  # a Washer has no set_heat
    except AttributeError as e:
        print("AttributeError:", e)
    print()

    print("Washer MRO:", [k.__name__ for k in Washer.__mro__])
    for name in ("start", "describe", "label", "set_heat"):
        print(f"  Washer.{name:8} found in {where_is(Washer, name)}")


if __name__ == "__main__":
    main()
```

```
W-01: fill water, spin at 1200 rpm
m.label = washer
m.describe() = W-01 is a washer

W-02: fill water, spin at 1200 rpm
D-01: tumble with heat level 2
M-00: generic machine starts

D-01: heat set to 3
AttributeError: 'Washer' object has no attribute 'set_heat'

Washer MRO: ['Washer', 'Machine', 'object']
  Washer.start    found in Washer
  Washer.describe found in Machine
  Washer.label    found in Washer
  Washer.set_heat found in None
```

When Python evaluates `m.start`, it performs a lookup that the helper `where_is` reproduces in a few lines. Every class has a dictionary, `__dict__`, mapping the names defined *in that class* to their values, and a tuple, `__mro__` (method resolution order), listing the class, its superclass, and so on up to `object`. Python first checks the object's own dictionary (for methods this finds nothing), then walks `type(m).__mro__` and takes the first class dictionary that contains the name. For `Washer` the order is `Washer`, `Machine`, `object`; `start` is found in `Washer`, `describe` in `Machine`. There are no slot numbers and no copied tables: the "table" is the chain of dictionaries, searched in order, with a cache so that the search is rarely repeated.

Two consequences differ from Java. First, `m.label` printed `washer`: in Python `label` is a *class attribute*, found by the same MRO walk as methods, so the subclass's value wins and `describe()` sees it too. Python has no hidden-field trap. Second, nothing is checked before the program runs: `floor[0].set_heat(1)` fails only when it executes, with `AttributeError`. `isinstance(x, Dryer)`

plays the role of Java's `instanceof`, but no cast follows it: `x.set_heat(3)` works because the object has the method. The annotation `m: Machine` is a *hint* for readers and checking tools; the interpreter ignores it.

> ✓ **KEY INSIGHT.** Java answers "may I call this?" at compile time from the *static* type, and "which version runs?" at run time from the *dynamic* type. Python answers both questions at run time from the object alone. Java catches more mistakes before the program runs; Python lets you write less. Neither answer changes the central fact: **the object, not the code that calls it, decides which method runs**.

7.8 The Root of Every Hierarchy: `Object` and `object`

Every Java class that does not say `extends` extends `java.lang.Object`, and so, directly or indirectly, *every* class is a subclass of `Object`. In Python every class inherits from `object`. That is why `Animal` could override `toString()` without inheriting from anything visible, and why the vtables of **Figure 7.4** end with `Object.toString`. The root classes provide a handful of methods that every object has; three of them matter every day, and all three are meant to be overridden:

- **Text.** Java's `toString()` and Python's `__str__()` (for users) and `__repr__()` (for programmers) produce the text shown when an object is printed. The inherited versions print the class name and an identity number, such as `PlainCustomer@1b6d3586`; useful to the machine, useless to a person.
- **Equality.** Java's `equals(Object)` and Python's `__eq__` decide whether two objects are *equal*. The inherited versions say "equal only if they are the very same object", which Java also offers as `==` and Python as `is`.
- **Hashing.** Java's `hashCode()` and Python's `__hash__` return an integer used by hash tables (`HashSet`, `HashMap`, Python's `set` and `dict`) to decide where to store an object. They must agree with equality.

Listing 7.11 shows why the defaults are usually wrong for a class that describes a real-world thing, such as a laundry customer identified by a phone number.

Listing 7.11 `ObjectRoot.java`: a class that inherits `toString`, `equals` and `hashCode` from `Object`, and one that overrides all three.

```
import java.util.HashSet;
import java.util.Objects;
import java.util.Set;

public class ObjectRoot {
    public static void main(String[] args) {
        PlainCustomer p1 = new PlainCustomer("Siti Aminah", "081234567890");
        PlainCustomer p2 = new PlainCustomer("Siti Aminah", "081234567890");
        System.out.println("PlainCustomer text starts with 'PlainCustomer@'? "
                + p1.toString().startsWith("PlainCustomer@"));
        System.out.println("p1.equals(p2) = " + p1.equals(p2));

        Customer c1 = new Customer("Siti Aminah", "081234567890");
        Customer c2 = new Customer("Siti A.", "081234567890");  // same phone
        Customer c3 = new Customer("Budi Santoso", "085712340000");
        System.out.println(c1);
        System.out.println("c1.equals(c2) = " + c1.equals(c2));
        System.out.println("c1 == c2      = " + (c1 == c2));

        Set<PlainCustomer> plain = new HashSet<>();
        plain.add(p1);
        plain.add(p2);
        Set<Customer> real = new HashSet<>();
        real.add(c1);
        real.add(c2);
        real.add(c3);
        System.out.println("distinct plain customers: " + plain.size());
        System.out.println("distinct customers:       " + real.size());
    }
}

class PlainCustomer {  // inherits toString, equals, hashCode from Object
    private final String name;
    private final String phone;

    PlainCustomer(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }
}

class Customer {
    private final String name;
    private final String phone;  // the phone number identifies a customer
```

Listing 7.11 (Continued) *ObjectRoot.java: a class that inherits toString, equals and hashCode from Object, and one that overrides all three.*

```
    Customer(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }

    @Override
    public String toString() {
        return "Customer[" + name + ", " + phone + "]";
    }

    @Override
    public boolean equals(Object other) {
        if (this == other) {
            return true;
        }
        if (!(other instanceof Customer that)) {
            return false;
        }
        return phone.equals(that.phone);
    }

    @Override
    public int hashCode() {  // equal objects MUST have equal hash codes
        return Objects.hash(phone);
    }
}
```

```
PlainCustomer text starts with 'PlainCustomer@'? true
p1.equals(p2) = false
Customer[Siti Aminah, 081234567890]
c1.equals(c2) = true
c1 == c2      = false
distinct plain customers: 2
distinct customers:       2
```

`PlainCustomer` overrides nothing. Two objects built from the same name and phone are not `equals`, so a `HashSet` keeps both: the laundry would count Siti twice. (We print only whether the text *starts with* `PlainCustomer@`, because the number after the `@` differs on every run.) `Customer` overrides all three. Its `equals` follows a pattern worth memorizing: `true` at once if the other object is this very object; `false` if it is not a `Customer` at all (the `instanceof` pattern also gives us `that`, typed as `Customer`); otherwise compare the fields that define identity, here the phone. Its `hashCode` hashes exactly the same field. Now `c1` and `c2` are equal but not the same object (`==` is `false`), and the set of three customers has two elements.

> ⚠ **WARNING.** If you override `equals`, you **must** override `hashCode` so that equal objects have equal hash codes. Java does not check this. A class that breaks the rule compiles and runs, and then a `HashSet` silently keeps "duplicates" and a `HashMap` silently fails to find keys that are there [6].

Python makes the rule harder to break, as **Listing 7.12** shows.

Listing 7.12 `object_root.py`: the same three methods in Python, and a class that defines `__eq__` without `__hash__`.

```
"""object_root.py: every class inherits from object; override __str__, __eq__, __hash__."""


class PlainCustomer:  # inherits __repr__, __eq__ and __hash__ from object
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone


class Customer:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone  # the phone number identifies a customer

    def __str__(self):
        return f"Customer[{self.name}, {self.phone}]"

    def __eq__(self, other):
        if not isinstance(other, Customer):
            return NotImplemented
        return self.phone == other.phone

    def __hash__(self):  # equal objects MUST have equal hashes
        return hash(self.phone)


class HalfDoneCustomer:
    def __init__(self, phone):
        self.phone = phone

    def __eq__(self, other):  # __eq__ without __hash__ ...
        return isinstance(other, HalfDoneCustomer) and self.phone == other.phone


def main():
    p1 = PlainCustomer("Siti Aminah", "081234567890")
    p2 = PlainCustomer("Siti Aminah", "081234567890")
    print("PlainCustomer text starts with '<__main__.PlainCustomer object'?",
          repr(p1).startswith("<__main__.PlainCustomer object"))
    print("p1 == p2 =", p1 == p2)
```

Listing 7.12 (Continued) *object_root.py: the same three methods in Python, and a class that defines __eq__ without __hash__.*

```
    c1 = Customer("Siti Aminah", "081234567890")
    c2 = Customer("Siti A.", "081234567890")  # same phone
    c3 = Customer("Budi Santoso", "085712340000")
    print(c1)
    print("c1 == c2 =", c1 == c2)
    print("c1 is c2 =", c1 is c2)
    print("distinct plain customers:", len({p1, p2}))
    print("distinct customers:      ", len({c1, c2, c3}))
    try:
        {HalfDoneCustomer("0812")}
    except TypeError as e:
        print("TypeError:", e)  # ... makes the class unhashable


if __name__ == "__main__":
    main()
```

```
PlainCustomer text starts with '<__main__.PlainCustomer object'? True
p1 == p2 = False
Customer[Siti Aminah, 081234567890]
c1 == c2 = True
c1 is c2 = False
distinct plain customers: 2
distinct customers:       2
TypeError: unhashable type: 'HalfDoneCustomer'
```

The Python code is shorter but follows the same pattern. `__eq__` returns the special value `NotImplemented`, rather than `False`, when the other object is of an unrelated type; this politely lets Python ask the other object whether *it* knows how to compare. The last class shows Python's safety net: a class that defines `__eq__` but not `__hash__` has its `__hash__` set to `None` automatically, so its objects cannot go into a set at all. A loud `TypeError` is far better than Java's silent misbehavior.

7.9 Abstract Classes

We have been using abstract classes since **Listing 7.1**. It is time to state the idea properly and see what it buys.

> **DEFINITION.** An **abstract method** is a method declared without a body. An **abstract class** is a class that cannot be instantiated; in Java it is marked `abstract`, and it must be abstract if it declares or inherits any abstract method it does not implement. A **concrete class** is one that can be instantiated; it must implement every abstract method it inherits.

An abstract class sits between an interface, which (traditionally) has only method declarations, and a concrete class, which has only complete methods. It can contain fields, constructors, complete methods and abstract methods all at once. Its purpose is to capture *what a family of classes has in common*, including code, while leaving *the parts that differ* to the subclasses. There are no `DB` objects — `new DB()` does not compile — but there are `LinearDB` and `TreeDB` objects, and variables of type `DB` can refer to either.

A hierarchy may have several abstract levels. Consider geometric figures. Every figure has a length (a line's length, a closed figure's perimeter). Some figures are *open* — a line, a curve — and enclose no area; others are *closed* — a rectangle, an oval — and do. The left tree of **Figure 7.5** draws the hierarchy; the right tree is a similar hierarchy for students, which Exercise 7.4 asks you to implement.

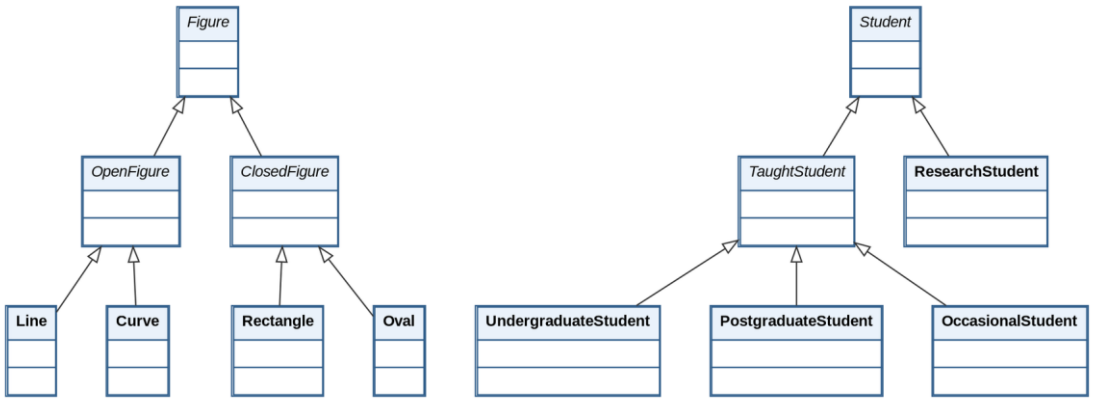


Figure 7.5 Two hierarchies with abstract inner levels (names in italics). Left: geometric figures, split into open and closed figures. Right: students, split into taught and research students.

Listing 7.13 implements the figures. The two concrete closed figures need two formulas each. A rectangle of width w and height h has perimeter $2(w + h)$ and area wh. An oval (an ellipse) with semi-axes a and b — half its width and half its height — has area

$$A = \pi ab \tag{7.2}$$

but its perimeter has no exact formula in elementary functions. A beautiful approximation found by the mathematician Srinivasa Ramanujan in 1914 is

$$P \approx \pi\left[3(a + b) - \sqrt{(3a + b)(a + 3b)}\right] \tag{7.3}$$

which is exact for a circle ($a = b = r$ gives $\pi\left[6r - \sqrt{16r^2}\right] = 2\pi r$) and within a fraction of a percent for any oval you are likely to draw.

Listing 7.13 `FigureDemo.java`: an abstract `Figure` with abstract subclasses `OpenFigure` and `ClosedFigure` and four concrete figures.

```
import java.util.Locale;

public class FigureDemo {
    public static void main(String[] args) {
        Figure[] figures = {
            new Line(0, 0, 3, 4),
            new Curve(new double[] {0, 1, 2}, new double[] {0, 1, 0}),
            new Rectangle(4, 2.5),
            new Oval(3, 2)
        };
        for (Figure f : figures) {
            System.out.println(f.describe());
        }
        double totalArea = 0;
        for (Figure f : figures) {
            if (f instanceof ClosedFigure c) {
                totalArea += c.area();
            }
        }
        System.out.printf(Locale.US, "Total area of the closed figures: %.3f%n",
totalArea);
    }
}

abstract class Figure {
    private final String name;

    protected Figure(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    abstract double length();  // every figure has a length (or perimeter)

    // A concrete method built on an abstract one.
    String describe() {
        return String.format(Locale.US, "%-9s length %7.3f", name, length());
    }
}
```

Listing 7.13 (Continued) *FigureDemo.java: an abstract Figure with abstract subclasses OpenFigure and ClosedFigure and four concrete figures.*

```
abstract class OpenFigure extends Figure {
    protected OpenFigure(String name) {
        super(name);
    }
}

abstract class ClosedFigure extends Figure {
    protected ClosedFigure(String name) {
        super(name);
    }

    abstract double area();  // only closed figures enclose an area

    @Override
    String describe() {
        return super.describe() + String.format(Locale.US, "   area %7.3f", area());
    }
}

class Line extends OpenFigure {
    private final double x1, y1, x2, y2;

    Line(double x1, double y1, double x2, double y2) {
        super("Line");
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }

    @Override
    double length() {
        return Math.hypot(x2 - x1, y2 - y1);
    }
}

class Curve extends OpenFigure {
    private final double[] xs, ys;  // a curve drawn as a polyline through points

    Curve(double[] xs, double[] ys) {
        super("Curve");
        this.xs = xs;
        this.ys = ys;
    }
```

Listing 7.13 (Continued) *FigureDemo.java: an abstract Figure with abstract subclasses OpenFigure and ClosedFigure and four concrete figures.*

```
    @Override
    double length() {
        double sum = 0;
        for (int i = 1; i < xs.length; i++) {
            sum += Math.hypot(xs[i] - xs[i - 1], ys[i] - ys[i - 1]);
        }
        return sum;
    }
}

class Rectangle extends ClosedFigure {
    private final double width, height;

    Rectangle(double width, double height) {
        super("Rectangle");
        this.width = width;
        this.height = height;
    }

    @Override
    double length() {
        return 2 * (width + height);
    }

    @Override
    double area() {
        return width * height;
    }
}

class Oval extends ClosedFigure {
    private final double a, b;  // the two semi-axes

    Oval(double a, double b) {
        super("Oval");
        this.a = a;
        this.b = b;
    }

    @Override
    double length() {  // Ramanujan's approximation of the perimeter
        return Math.PI * (3 * (a + b) - Math.sqrt((3 * a + b) * (a + 3 * b)));
    }
```

Listing 7.13 (Continued) *FigureDemo.java: an abstract Figure with abstract subclasses OpenFigure and ClosedFigure and four concrete figures.*

```
    @Override
    double area() {
        return Math.PI * a * b;
    }
}
```

```
Line      length   5.000
Curve     length   2.828
Rectangle length  13.000   area  10.000
Oval      length  15.865   area  18.850
Total area of the closed figures: 28.850
```

The design deserves a careful look, because it shows the three things an abstract class can do.

1. **Declare what every subclass must provide.** `Figure` declares `abstract double length()`. Every concrete figure must supply it, and the compiler checks: a concrete class that forgets an inherited abstract method does not compile.
2. **Provide code that relies on what subclasses provide.** `Figure.describe()` is complete. It calls `length()` without knowing which figure it is measuring, and dynamic dispatch supplies the right formula. A method like this — a fixed outline with a step filled in by subclasses — is called a **template method**, one of the classic design patterns [7].
3. **Add a requirement for part of the family.** `ClosedFigure` is also abstract. It adds `abstract double area()`, which only closed figures have, and overrides `describe()` to append the area, calling `super.describe()` for the part that `Figure` already knows how to write.

Check the numbers. The line from (0,0) to (3,4) has length $\sqrt{3^2 + 4^2} = 5$; `Math.hypot` computes exactly this. The curve through (0,0), (1,1), (2,0) is two segments of length $\sqrt{2}$, total $2\sqrt{2} \approx 2.828$. The rectangle has perimeter $2(4 + 2.5) = 13$ and area 10. For the oval, Equation (7.2) gives $\pi \cdot 3 \cdot 2 \approx 18.850$ and Equation (7.3) gives $\pi\left[15 - \sqrt{11 \cdot 9}\right] = \pi(15 - 9.950) \approx 15.865$. The last line sums the areas of the closed figures only, using `instanceof ClosedFigure` because `area()` does not exist for open figures: $10 + 18.850 = 28.850$.

One detail deserves a word: `String.format` and `printf` follow the computer's regional settings, so on a computer set to Indonesian `%.3f` prints `5,000` with a decimal comma. Passing `Locale.US` makes the output the same everywhere; an application for Indonesian customers would pass `Locale.forLanguageTag("id-ID")`.

What does the compiler say if we try to create a plain figure, or forget a method? Adding `Figure plain = new Figure("?");` at the start of `main` gives

```
FigureDemo.java:5: error: Figure is abstract; cannot be instantiated
        Figure plain = new Figure("?");
                       ^
1 error
```

and a concrete subclass that forgets an abstract method is rejected with a message naming the missing method; we will see one in Section 7.15.

Listing 7.14 is the Python version.

Listing 7.14 `figure_demo.py`: the figure hierarchy in Python; `OpenFigure` and `ClosedFigure` list `ABC` again to make clear that they, too, are abstract.

```python
"""figure_demo.py: an abstract Figure hierarchy with open and closed figures."""
import math
from abc import ABC, abstractmethod


class Figure(ABC):
    def __init__(self, name):
        self._name = name

    @abstractmethod
    def length(self):  # every figure has a length (or perimeter)
        ...

    # A concrete method built on an abstract one.
    def describe(self):
        return f"{self._name:<9} length {self.length():7.3f}"


class OpenFigure(Figure, ABC):
    pass


class ClosedFigure(Figure, ABC):
    @abstractmethod
    def area(self):  # only closed figures enclose an area
        ...

    def describe(self):
        return super().describe() + f"   area {self.area():7.3f}"


class Line(OpenFigure):
    def __init__(self, x1, y1, x2, y2):
        super().__init__("Line")
        self._p, self._q = (x1, y1), (x2, y2)

    def length(self):
        return math.dist(self._p, self._q)


class Curve(OpenFigure):
    def __init__(self, xs, ys):  # a curve drawn as a polyline through points
        super().__init__("Curve")
        self._points = list(zip(xs, ys))
```

Listing 7.14 (Continued) *figure_demo.py: the figure hierarchy in Python; OpenFigure and ClosedFigure list ABC again to make clear that they, too, are abstract.*

```
    def length(self):
        return sum(math.dist(a, b) for a, b in zip(self._points, self._points[1:]))


class Rectangle(ClosedFigure):
    def __init__(self, width, height):
        super().__init__("Rectangle")
        self._w, self._h = width, height

    def length(self):
        return 2 * (self._w + self._h)

    def area(self):
        return self._w * self._h


class Oval(ClosedFigure):
    def __init__(self, a, b):  # the two semi-axes
        super().__init__("Oval")
        self._a, self._b = a, b

    def length(self):  # Ramanujan's approximation of the perimeter
        a, b = self._a, self._b
        return math.pi * (3 * (a + b) - math.sqrt((3 * a + b) * (a + 3 * b)))

    def area(self):
        return math.pi * self._a * self._b


def main():
    figures = [Line(0, 0, 3, 4), Curve([0, 1, 2], [0, 1, 0]), Rectangle(4, 2.5), Oval(3,
2)]
    for f in figures:
        print(f.describe())
    total_area = sum(f.area() for f in figures if isinstance(f, ClosedFigure))
    print(f"Total area of the closed figures: {total_area:.3f}")
    try:
        Figure("?")
    except TypeError as e:
        print("Figure():", e)


if __name__ == "__main__":
    main()
```

```
Line      length   5.000
Curve     length   2.828
Rectangle length  13.000   area  10.000
Oval      length  15.865   area  18.850
Total area of the closed figures: 28.850
Figure(): Can't instantiate abstract class Figure with abstract method length
```

`math.dist` computes the distance between two points, playing the role of `Math.hypot`. `Curve.length` pairs each point with the next using `zip(self._points, self._points[1:])`, and sums the distances. The final `sum(...)` with an `if` inside is a *generator expression*, Python's compact form of "loop, filter, add up". Python checks abstract methods at the moment an object is created, and the last line of output shows the refusal.

7.9.1 The Java library is built the same way

Abstract classes are not an academic device. **Figure 7.6** shows a corner of Java's own collection library, which Chapter 5 used and Chapter 9 studies in depth.

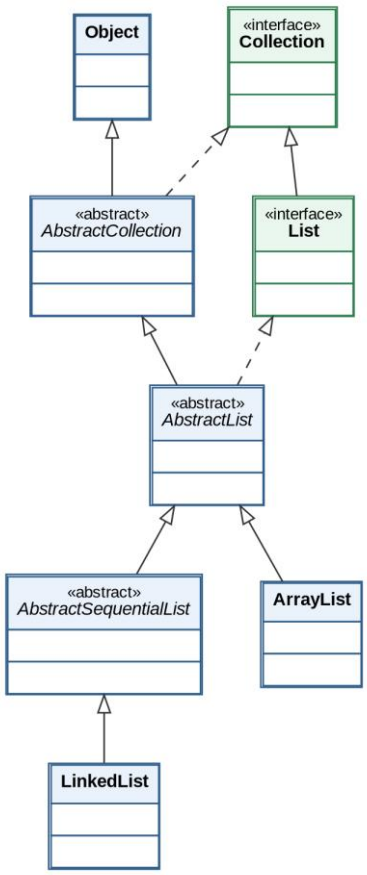


Figure 7.6 Part of the Java collections hierarchy. Solid lines with hollow triangles are `extends`; dashed lines are `implements`. The abstract classes supply shared code, the interfaces state the contracts, and `ArrayList` and `LinkedList` are the concrete classes programs actually create.

`Collection` and `List` are interfaces: they promise methods such as `add`, `get` and `size`. `AbstractCollection` and `AbstractList` are abstract classes that implement most of those methods in terms of a few that they leave abstract — exactly as `DB` implements `findOneOf` in terms of `search`. `AbstractList`, for instance, implements `contains`, `indexOf`, `iterator` and many more in terms of just `get(int)` and `size()`. `ArrayList` extends `AbstractList` and supplies an array; `LinkedList` extends `AbstractSequentialList` and supplies a chain of nodes. A programmer who wants a new kind of list writes two methods and inherits dozens.

7.10 Inheriting from Concrete Classes, and Why to Be Careful

Nothing forces a superclass to be abstract. A class may extend an ordinary, *concrete* class, and the mechanism is exactly the same: the subclass gets every field and method and may override any of them. The only difference is that the superclass can have objects of its own. So we could skip `DB` altogether and let `TreeDB` inherit from `LinearDB`, as **Listing 7.15** does. It lives in a package `dbconcrete` so that its class names do not clash with those of the `db` package.

Listing 7.15 `dbconcrete/DBTest.java`: `TreeDB` inherits from the concrete class `LinearDB`.

```
package dbconcrete;

public class DBTest {
    public static void main(String[] args) {
        LinearDB ldb = new LinearDB();
        TreeDB tdb = new TreeDB();
        ldb.addKey(1);
        if (ldb.search(1)) {
            System.out.println("It's found!");
        } else {
            System.out.println("It's not found!");
        }
        tdb.addKey(2);
        if (tdb.search(1)) {
            System.out.println("It's found!");
        } else {
            System.out.println("It's not found!");
        }
        // What TreeDB received from LinearDB without asking for it:
        System.out.println("-- what TreeDB inherited --");
        System.out.println("tdb.table.length = " + tdb.table.length);
        System.out.println("tdb is a LinearDB? " + (tdb instanceof LinearDB));
        LinearDB surprise = new TreeDB();  // legal, but is a tree "a linear DB"?
        surprise.addSeveral(new int[] {7, 8});
    }
}

class LinearDB {
    int table[] = new int[10];
    int size = 0;

    void addKey(int key) {
        System.out.println("LinearDB: addKey()");
    }

    boolean search(int key) {
        boolean result = true;
        System.out.println("LinearDB: searchKey()");
        return result;
    }
```

Listing 7.15 (Continued) *dbconcrete/DBTest.java: TreeDB inherits from the concrete class LinearDB.*

```
    void addSeveral(int keys[]) {
        for (int i = 0; i < keys.length; i++) {
            addKey(keys[i]);
        }
    }

    boolean findOneOf(int keys[]) {
        for (int i = 0; i < keys.length; i++) {
            if (search(keys[i])) {
                return true;
            }
        }
        return false;
    }
}

class TreeDB extends LinearDB {
    Object data;  // a binary search tree would live here

    void addKey(int key) {
        System.out.println("TreeDB: addKey()");
    }

    boolean search(int key) {
        boolean result = false;
        System.out.println("TreeDB: searchKey()");
        return result;
    }
}
```

```
LinearDB: addKey()
LinearDB: searchKey()
It's found!
TreeDB: addKey()
TreeDB: searchKey()
It's not found!
-- what TreeDB inherited --
tdb.table.length = 10
tdb is a LinearDB? true
TreeDB: addKey()
TreeDB: addKey()
```

It works; the first six lines are the same as before. `addSeveral` and `findOneOf` are written once, in `LinearDB`, and `TreeDB` inherits them. So why be careful? The last block of output answers, and **Figure 7.7** draws the answer.

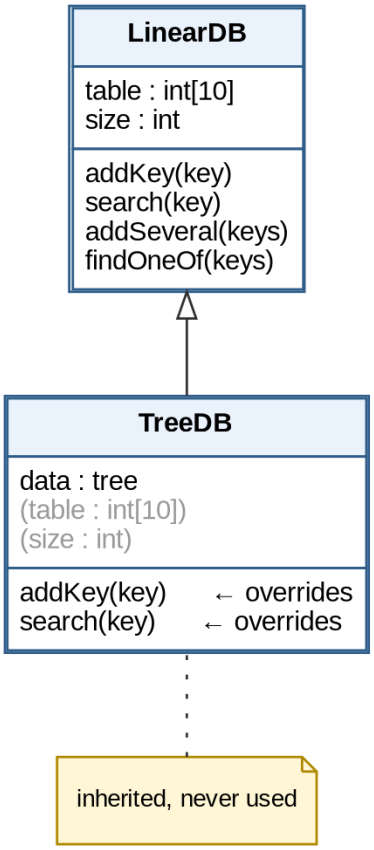


Figure 7.7 Inheriting from a concrete class. `TreeDB` receives `LinearDB`'s array and counter although a tree never uses them, and every `TreeDB` now claims to be a `LinearDB`.

- **Baggage.** Every `TreeDB` object carries an array of ten integers and a `size` counter that it never uses: `tdb.table.length = 10`. Here the waste is small. In real hierarchies it can be database connections, open files, or caches.
- **A false promise.** `tdb instanceof LinearDB` is `true`, and `LinearDB surprise = new TreeDB();` compiles. Any code written for "a linear database" — which might reasonably assume, say, that keys come back in insertion order — will now accept a tree and be silently wrong. A tree *is not* a linear database. The is-a test of Section 7.2 fails, and the compiler cannot notice.
- **Coupling to details.** `TreeDB` now depends on everything `LinearDB` does, including things `LinearDB` may change tomorrow. Section 7.11 shows how such a change can break a subclass that did nothing wrong.

The version with an abstract `DB` has none of these problems: `DB` holds only what *every* database shares, and `LinearDB` and `TreeDB` are siblings, neither pretending to be the other. That is the general lesson: **when two classes share code, extract an abstract superclass that both extend, rather than making one extend the other.**

Listing 7.16 repeats the experiment in Python. Note the comment in `TreeDB.__init__`: because Python never calls the superclass initializer by itself, a Python `TreeDB` that forgot `super().__init__()` would *not* have the array — the baggage appears only if we ask for it, and the inherited `add_several` would still work, since it never touches the array. The false promise, however, is there either way.

Listing 7.16 `db_concrete.py`: the concrete-superclass version in Python.

```
"""db_concrete.py: TreeDB inherits from the concrete class LinearDB."""


class LinearDB:
    def __init__(self):
        self.table = [0] * 10
        self.size = 0

    def add_key(self, key):
        print("LinearDB: addKey()")

    def search(self, key):
        result = True
        print("LinearDB: searchKey()")
        return result

    def add_several(self, keys):
        for key in keys:
            self.add_key(key)

    def find_one_of(self, keys):
        for key in keys:
            if self.search(key):
                return True
        return False


class TreeDB(LinearDB):
    def __init__(self):
        super().__init__()  # we must call it, or table and size never exist
        self.data = None    # a binary search tree would live here

    def add_key(self, key):
        print("TreeDB: addKey()")

    def search(self, key):
        result = False
        print("TreeDB: searchKey()")
        return result
```

Listing 7.16 (Continued) *db_concrete.py: the concrete-superclass version in Python.*

```
def main():
    ldb = LinearDB()
    tdb = TreeDB()
    ldb.add_key(1)
    if ldb.search(1):
        print("It's found!")
    else:
        print("It's not found!")
    tdb.add_key(2)
    if tdb.search(1):
        print("It's found!")
    else:
        print("It's not found!")
    # What TreeDB received from LinearDB without asking for it:
    print("-- what TreeDB inherited --")
    print("len(tdb.table) =", len(tdb.table))
    print("tdb is a LinearDB?", isinstance(tdb, LinearDB))
    surprise: LinearDB = TreeDB()  # legal, but is a tree "a linear DB"?
    surprise.add_several([7, 8])


if __name__ == "__main__":
    main()
```

```
LinearDB: addKey()
LinearDB: searchKey()
It's found!
TreeDB: addKey()
TreeDB: searchKey()
It's not found!
-- what TreeDB inherited --
len(tdb.table) = 10
tdb is a LinearDB? True
TreeDB: addKey()
TreeDB: addKey()
```

7.11 Composition versus Inheritance

Inheritance is not the only way to reuse code. The other way is to *use* an object of another class rather than to *be* one: a class keeps a reference to a helper object in a field and calls its methods. This is called **composition** (or, when the helper can exist independently, *aggregation*), and it expresses the relationship **has-a**: an `Order` has services; a car has an engine.

> **DEFINITION. Inheritance** reuses code through an *is-a* relationship: the subclass *is* a kind of the superclass and receives its members. **Composition** reuses code through a *has-a* relationship: the class *contains* an object of another class and delegates work to it by calling its public methods.

The advice of almost every experienced designer, first written down in the book that named the classic design patterns, is: **favor object composition over class inheritance** [7]. Inheritance is not bad, but it creates the tightest coupling there is between two classes: a subclass depends not only on what its superclass *promises* but on *how it is implemented*. Alan Snyder, analyzing the first generation of object-oriented languages, observed that inheritance lets a subclass depend on the superclass's internal choices, and so weakens the encapsulation that objects are meant to provide [8].

7.11.1 The fragile base class

Here is the classic demonstration, adapted from Bloch [6]. The laundry wants a set of garment tags (`T-101`, `T-102`, ...) that also counts how many tags were ever added. The obvious approach extends `HashSet<String>` and overrides the two adding methods to increment a counter. **Listing 7.17** writes that, and next to it the same class built by composition.

Listing 7.17 `FragileBase.java`: a counting tag set built by inheritance, which counts every tag twice, and one built by composition, which counts correctly.

```
import java.util.Collection;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class FragileBase {
    public static void main(String[] args) {
        List<String> tags = List.of("T-101", "T-102", "T-103");

        InheritedTagSet inherited = new InheritedTagSet();
        inherited.addAll(tags);
        System.out.println("inherited: size = " + inherited.size()
                + ", tags added = " + inherited.getAddCount());

        ComposedTagSet composed = new ComposedTagSet();
        composed.addAll(tags);
        System.out.println("composed:  size = " + composed.size()
                + ", tags added = " + composed.getAddCount());
    }
}

// Inheritance: IS-A HashSet, and depends on how HashSet works inside.
class InheritedTagSet extends HashSet<String> {
    private int addCount = 0;

    @Override
    public boolean add(String tag) {
        addCount++;
        return super.add(tag);
    }

    @Override
    public boolean addAll(Collection<? extends String> tags) {
        addCount += tags.size();
        return super.addAll(tags);  // HashSet.addAll calls add() for each tag!
    }

    int getAddCount() {
        return addCount;
    }
}
```

Listing 7.17 (Continued) *FragileBase.java: a counting tag set built by inheritance, which counts every tag twice, and one built by composition, which counts correctly.*

```
// Composition: HAS-A set, and only uses its public contract.
class ComposedTagSet {
    private final Set<String> tags = new HashSet<>();
    private int addCount = 0;

    boolean add(String tag) {
        addCount++;
        return tags.add(tag);
    }

    boolean addAll(Collection<? extends String> more) {
        addCount += more.size();
        return tags.addAll(more);  // the inner set's add() is not ours: no double count
    }

    int size() {
        return tags.size();
    }

    int getAddCount() {
        return addCount;
    }
}
```

```
inherited: size = 3, tags added = 6
composed:  size = 3, tags added = 3
```

We added three tags, and the inherited version says six. Nothing in `InheritedTagSet` looks wrong; each method adds the right amount. The problem is hidden in `HashSet`: its `addAll`, inherited from `AbstractCollection`, is implemented by calling `add` once per element. Follow **Figure 7.8**, left side. Our `addAll` adds 3 to the counter, then calls `super.addAll`, which calls `add` three times — and because of dynamic dispatch those calls go to *our* overriding `add`, which adds 1 each time. Total: six.

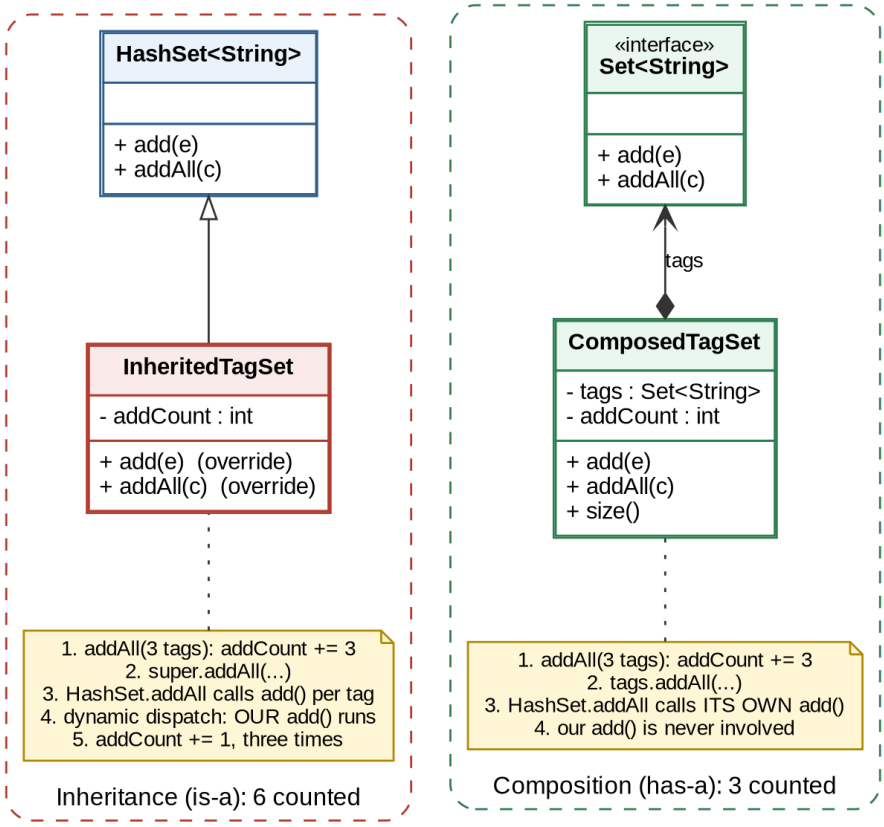


Figure 7.8 The fragile base class. Left: the subclass's `addAll` calls the superclass's `addAll`, which calls `add`, which dynamic dispatch sends back to the subclass: every tag is counted twice. Right: the composed class calls methods of an object it merely holds; its own `add` is never re-entered.

The frightening part is that whether `addAll` calls `add` is an *implementation detail* of `HashSet`. It is not in the documentation of `Set`; a future Java version could change it, and a subclass that was correct yesterday would break without a single line of it changing. Hence the name: the base class is *fragile*, because its subclasses depend on its insides.

`ComposedTagSet`, on the right of **Figure 7.8**, keeps a `Set<String>` in a private field and forwards calls to it. Its `addAll` calls `tags.addAll(more)`, which calls the inner set's own `add`; our class is not a `HashSet`, so dispatch cannot bounce back into it, and the count is 3. The composed class also exposes only the methods it chooses, while the inherited one exposes every method of `HashSet`, including `remove` and `clear`.

Python has exactly the same trap, and **Listing 7.18** springs it with `collections.UserDict`, a dictionary class designed to be subclassed.

Listing 7.18 `fragile_base.py`: the fragile base class in Python; `UserDict.update` calls `__setitem__` for every pair.

```
"""fragile_base.py: counting tags by inheriting from UserDict, and by composition."""
from collections import UserDict


# Inheritance: IS-A UserDict, and depends on how UserDict works inside.
class InheritedTagRegistry(UserDict):
    def __init__(self):
        super().__init__()
        self.add_count = 0

    def __setitem__(self, tag, garment):
        self.add_count += 1
        super().__setitem__(tag, garment)

    def update(self, pairs):
        self.add_count += len(pairs)
        super().update(pairs)  # UserDict.update calls __setitem__ for each pair!


# Composition: HAS-A dict, and only uses its public contract.
class ComposedTagRegistry:
    def __init__(self):
        self._tags = {}
        self.add_count = 0

    def __setitem__(self, tag, garment):
        self.add_count += 1
        self._tags[tag] = garment

    def update(self, pairs):
        self.add_count += len(pairs)
        self._tags.update(pairs)  # the inner dict's methods are not ours: no double count

    def __len__(self):
        return len(self._tags)


def main():
    pairs = {"T-101": "shirt", "T-102": "towel", "T-103": "sock"}

    inherited = InheritedTagRegistry()
    inherited.update(pairs)
    print(f"inherited: size = {len(inherited)}, tags added = {inherited.add_count}")
```

Listing 7.18 (Continued) *fragile_base.py: the fragile base class in Python; UserDict.update calls __setitem__ for every pair.*

```
    composed = ComposedTagRegistry()
    composed.update(pairs)
    print(f"composed:  size = {len(composed)}, tags added = {composed.add_count}")


if __name__ == "__main__":
    main()
```

```
inherited: size = 3, tags added = 6
composed:  size = 3, tags added = 3
```

(The built-in `dict` happens *not* to route `update` through `__setitem__` — another undocumented detail you would have to know, which rather proves the point.)

7.11.2 When is inheritance right? The substitution principle

Composition is the safe default, but inheritance is exactly right when the subclass truly *is* a special case of the superclass — when every piece of code written for the superclass would still be correct if handed a subclass object. Barbara Liskov and Jeannette Wing turned that sentence into a precise rule, now called the **Liskov substitution principle** [9]:

$$\forall x{:}\ T.\ \phi(x)\ \Rightarrow\ \forall y{:}\ S.\ \phi(y) \qquad \text{whenever } S \text{ is a subtype of } T \tag{7.4}$$

Read Equation (7.4) as: "if some property ϕ can be proved about every object x of type T, then it must also hold for every object y of the subtype S". The properties that matter are the promises callers rely on. If callers of `Service.price()` rely on the price being a positive number of rupiah, then no subclass may return zero or a negative number. If callers of a "linear database" rely on insertion order, then a `TreeDB` may not pose as one — which is precisely what went wrong in Section 7.10.

In practice, three questions settle most cases:

1. **Does the is-a sentence sound right to a non-programmer?** "An express service is a service": yes. "A tag set is a hash set": only to a programmer.
2. **Would every caller of the superclass be happy with a subclass object?** If a subclass must refuse or ignore some inherited method, the answer is no.
3. **Do you want the subclass to change when the superclass's insides change?** If not, compose.

The laundry price engine of Section 7.15 uses both tools, each where it belongs. `WashService` *is a* `Service`, so it extends `Service`. An express order *is a* `Service` too — an order can contain it — but it *has a* service inside, the one it speeds up. So `ExpressService` extends `Service` **and** holds a `Service` in a field. That combination is a known design pattern, the *decorator* [7], and it lets express wrap any service at all, including ones not yet invented.

7.12 Multiple Inheritance: Python's MRO and Java's Interfaces

Could a class have two superclasses? A "wash and iron" service is, after all, both a washing service and an ironing service. Languages answer this question differently, and the difference is instructive.

7.12.1 Java: one superclass, many interfaces

Java allows a class to `extend` exactly **one** class. It may, however, `implement` any number of **interfaces**. Chapter 6 introduced interfaces as lists of method declarations; **Table 7.3** compares them with abstract classes.

Table 7.3 Abstract classes and interfaces in Java.

	Abstract class	Interface
Can declare abstract methods	Yes	Yes (all methods without a body are abstract)
Can contain method code	Yes	Only `default`, `static` and `private` methods (since Java 8 and 9)
Can hold instance fields (state)	Yes	No — only constants
Has constructors	Yes	No
How many can a class use	One (`extends`)	Many (`implements A, B, C`)
Expresses	"is a kind of", with shared code and state	"can do", a capability or contract

The restriction to one superclass is deliberate: the real danger of multiple inheritance lies in inheriting *state* and *code* along two paths. Interfaces carry no state, so implementing several is safe. But since Java 8 an interface may carry `default` methods with code, so the question returns in a smaller form: what if two interfaces supply a default method with the same signature? **Listing 7.19** tries it.

Listing 7.19 `DiamondDemo.java`: two interfaces with conflicting default methods; the implementing class must resolve the conflict, and a method inherited from a class beats any default.

```
public class DiamondDemo {
    public static void main(String[] args) {
        PremiumCare premium = new PremiumCare();
        System.out.println("PremiumCare: " + premium.finish());

        Foldable asFoldable = premium;    // one object, seen through two interfaces
        Steamable asSteamable = premium;
        System.out.println("as Foldable:  " + asFoldable.finish());
        System.out.println("as Steamable: " + asSteamable.finish());

        HotelCare hotel = new HotelCare();
        System.out.println("HotelCare:    " + hotel.finish());  // the class wins
    }
}

interface Foldable {
    default String finish() {
        return "folded";
    }
}

interface Steamable {
    default String finish() {
        return "steamed";
    }
}

// Two defaults with the same signature: the class MUST decide.
class PremiumCare implements Foldable, Steamable {
    @Override
    public String finish() {
        return Steamable.super.finish() + ", then " + Foldable.super.finish();
    }
}

class BasicCare {
    public String finish() {
        return "hung on a rack";
    }
}

// A method inherited from a class beats a default from an interface.
class HotelCare extends BasicCare implements Foldable {
}
```

```
PremiumCare: steamed, then folded
as Foldable:  steamed, then folded
as Steamable: steamed, then folded
HotelCare:    hung on a rack
```

`Foldable` and `Steamable` each offer a default `finish()`. `PremiumCare` implements both. Java refuses to guess which one it should get; if `PremiumCare` did not override `finish()`, the compiler would stop with:

```
DiamondDemo.java:29: error: types Foldable and Steamable are incompatible;
class PremiumCare implements Foldable, Steamable {
^
  class PremiumCare inherits unrelated defaults for finish() from types Foldable and
Steamable
1 error
```

So `PremiumCare` overrides `finish()`, and inside it uses the special syntax `Steamable.super.finish()` and `Foldable.super.finish()` to call each interface's default explicitly. Notice that the object answers the same whether it is seen as a `Foldable` or a

`Steamable`: dynamic dispatch always finds the class's override. `HotelCare` illustrates the other rule: it inherits `finish()` from the *class* `BasicCare` and a default `finish()` from the *interface* `Foldable`, and the class wins without any complaint. Java's three rules for defaults, from the Language Specification [3], are: (1) a method from a class beats a default from an interface; (2) a default from a more specific interface beats one from an interface it extends; (3) otherwise the class must override and choose.

7.12.2 Python: real multiple inheritance and the MRO

Python allows a class to list several superclasses, with both state and code. The problem that Java avoided must therefore be solved, and it is solved by a single, carefully designed ordering. **Figure 7.9** shows the classic *diamond*: `Washing` and `Ironing` both inherit from `Service`, and `WashAndIron` inherits from both.

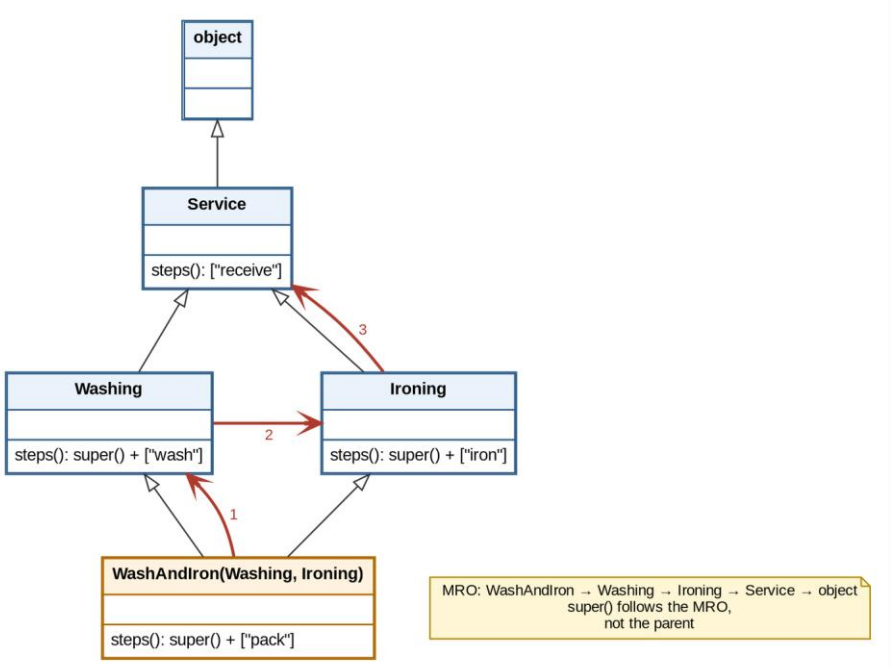


Figure 7.9 The diamond. `WashAndIron` has two paths up to `Service`. Python linearizes the hierarchy into one method resolution order, and every `super()` call moves to the next class in that order (numbered arrows), so `Service.steps` runs exactly once.

Listing 7.20 `diamond_mro.py`: a diamond hierarchy with cooperative `super()` calls, the resulting MRO, a conflict resolved by order, and a hierarchy for which no consistent order exists.

```
"""diamond_mro.py: multiple inheritance, the MRO, and cooperative super()."""


class Service:
    def steps(self):
        return ["receive"]


class Washing(Service):
    def steps(self):
        return super().steps() + ["wash"]


class Ironing(Service):
    def steps(self):
        return super().steps() + ["iron"]


class WashAndIron(Washing, Ironing):  # the diamond: two paths up to Service
    def steps(self):
        return super().steps() + ["pack"]


class Foldable:
    def finish(self):
        return "folded"


class Steamable:
    def finish(self):
        return "steamed"


class PremiumCare(Steamable, Foldable):  # no error: the MRO simply picks Steamable
    pass


def names(cls):
    return [k.__name__ for k in cls.__mro__]
```

Listing 7.20 (Continued) *diamond_mro.py: a diamond hierarchy with cooperative super() calls, the resulting MRO, a conflict resolved by order, and a hierarchy for which no consistent order exists.*

```
def main():
    print("WashAndIron MRO:", names(WashAndIron))
    print("steps():", WashAndIron().steps())
    print("PremiumCare MRO:", names(PremiumCare))
    print("finish():", PremiumCare().finish())
    try:
        class Confused(Service, Washing):  # asks for Service BEFORE its own subclass
            pass
    except TypeError as e:
        print("TypeError:", e)


if __name__ == "__main__":
    main()
```

```
WashAndIron MRO: ['WashAndIron', 'Washing', 'Ironing', 'Service', 'object']
steps(): ['receive', 'iron', 'wash', 'pack']
PremiumCare MRO: ['PremiumCare', 'Steamable', 'Foldable', 'object']
finish(): steamed
TypeError: Cannot create a consistent method resolution
order (MRO) for bases Service, Washing
```

Look at the second line of output. `WashAndIron().steps()` returned `['receive', 'iron', 'wash', 'pack']`. Where did `iron` come from? `WashAndIron.steps` calls `super().steps()`. Most beginners expect `super()` to mean "my parent", and would predict that `Washing.steps` runs, whose `super()` then reaches `Service`, giving `['receive', 'wash', 'pack']` — with ironing lost. That is not what happens. **In Python, `super()` means "the next class in the MRO of the object's class"**, not "my parent". The object is a `WashAndIron`, whose MRO is `WashAndIron, Washing, Ironing, Service, object`. So `WashAndIron.steps` calls `Washing.steps` (arrow 1 in **Figure 7.9**); `Washing`'s `super()` moves to the *next class after `Washing` in that MRO*, which is `Ironing` (arrow 2) — even though `Ironing` is not a superclass of `Washing`; `Ironing`'s `super()` moves to `Service` (arrow 3). `Service` returns `['receive']`, and the results are extended on the way back: `iron`, then `wash`, then `pack`. Every class runs once, and the shared ancestor runs once. This style, in which each class calls `super()` and trusts the MRO, is called **cooperative multiple inheritance** [10].

How is the MRO computed? Python uses the **C3 linearization**, designed for the language Dylan [11] and adopted by Python 2.3 [12]. Write $L[C]$ for the linearization (the MRO list) of class C, and suppose C is declared with bases $B_1, \ldots, B_n$ in that order. Then

$$L[C] = C + \text{merge}(L[B_1], \ldots, L[B_n],\ [B_1, \ldots, B_n]) \tag{7.5}$$

where merge repeatedly takes the first element (the *head*) of the first list whose head does not appear in the *tail* (any position but the first) of any other list, appends it to the result, and removes it from all lists. If no such head exists, the hierarchy has no consistent order.

Let us apply Equation (7.5) by hand to `WashAndIron(Washing, Ironing)`. First the smaller linearizations: L[Service] = [Service,object]; L[Washing] = [Washing,Service,object]; L[Ironing] = [Ironing,Service,object]. Now merge the three lists [Washing,Service,object], [Ironing,Service,object], [Washing,Ironing]:

1. Head of the first list is `Washing`. Is it in any tail? No. Take it. Lists: [Service,object], [Ironing,Service,object], [Ironing].
2. Head of the first list is `Service`. It is in the tail of the second list. Skip it; try the second list's head, `Ironing`. Not in any tail. Take it. Lists: [Service,object], [Service,object], [ ].
3. `Service` is no longer in any tail. Take it; then `object`.

Result: L[WashAndIron] = [WashAndIron,Washing,Ironing,Service,object] — exactly what the program printed. Step 2 is the heart of C3: `Service` must wait until *every* subclass of it has been listed, which is what guarantees that a shared ancestor comes after all its descendants.

The third and fourth lines of output show how Python settles the conflict that Java refused to settle. `PremiumCare(Steamable, Foldable)` inherits two `finish` methods; the MRO lists `Steamable` first because it was named first, so `Steamable.finish` wins. Python chooses silently, by order; Java makes you choose explicitly, which is harder to get wrong by accident.

The last line shows C3 refusing. `class Confused(Service, Washing)` asks for `Service` *before* `Washing`, but `Washing` is a subclass of `Service` and must come before it. The merge in step 2 finds no acceptable head, and Python raises `TypeError` at the moment the class statement runs. (The message is split over two lines in Python 3.11; newer versions print it on one.)

> ⚠ **WARNING.** Multiple inheritance in Python is powerful but easy to misuse. The safe pattern is a single main hierarchy plus small **mixin** classes — classes that add one capability, hold little or no state, and call `super()` cooperatively. When you are tempted to inherit from two "big" classes, it is usually a sign that one of them should be a field instead (Section 7.11).

7.13 Access Control

So far we have used `private`, `protected` and `public` without a full explanation. Inheritance makes the explanation urgent: a subclass is a new kind of "outsider" that sits somewhere between the class itself and the rest of the world, and we must decide what it may see.

7.13.1 Java's four access levels

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. Java has them at two levels:

- **At the top level** — for a class itself — a class may be `public` (usable from any package) or have no modifier, called **package-private** (usable only inside its own package). All the helper classes in this chapter's listings, such as `DB` and `LinearDB`, are package-private; only the class with `main` is public.
- **At the member level** — for fields, methods and constructors — there are four choices: `public`, `protected`, package-private (no modifier), and `private`.

Table 7.4 is the reference table, in the form given by Java's official tutorial [13]. A "Y" means that code in that place can access a member declared with that modifier.

Table 7.4 Java access levels. *Class*: code in the same class; *Package*: code in any class of the same package; *Subclass*: code in a subclass declared in another package; *World*: any other code.

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

The columns are easiest to understand with four concrete classes arranged in two packages, as in **Figure 7.10**: `Alpha` and `Beta` in package `one`, and `AlphaSub` (a subclass of `Alpha`) and `Gamma` in package `two`.

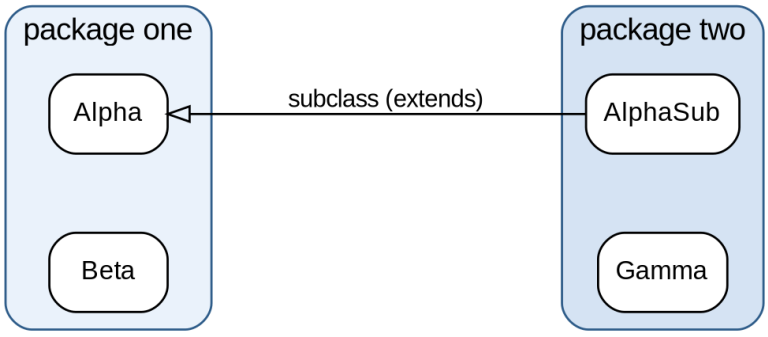


Figure 7.10 Four classes in two packages. `Beta` shares a package with `Alpha`; `AlphaSub` is in another package but extends `Alpha`; `Gamma` has no relationship with `Alpha` at all.

Reading **Table 7.4** from `Alpha`'s point of view gives **Table 7.5**: which of `Alpha`'s members each of the four classes can see.

Table 7.5 Visibility of `Alpha`'s members from each class of **Figure 7.10**.

Modifier	Alpha	Beta	AlphaSub	Gamma
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Tables are easy to memorize and easy to misremember, so let us make the compiler prove them. **Listing 7.21**, **Listing 7.22**, **Listing 7.23** and **Listing 7.24** are the four classes, each in its own file, because a `public` class must live in a file of the same name, and a package is a folder.

Listing 7.21 `one/Alpha.java`: one field of each access level.

```
package one;

public class Alpha {
    public String publicField = "public";
    protected String protectedField = "protected";
    String packageField = "package";  // no modifier: package-private
    private String privateField = "private";

    public String allFour() {  // inside Alpha every member is visible
        return publicField + " " + protectedField + " " + packageField + " " +
privateField;
    }
}
```

Listing 7.22 `one/Beta.java`: same package as `Alpha`, not a subclass.

```
package one;

public class Beta {  // same package as Alpha, not a subclass
    public static String look(Alpha a) {
        return a.publicField + " " + a.protectedField + " " + a.packageField;
        // a.privateField would not compile: private to Alpha
    }
}
```

Listing 7.23 `two/AlphaSub.java`: another package, but a subclass of `Alpha`.

```
package two;

import one.Alpha;

public class AlphaSub extends Alpha {  // other package, but a subclass
    public String look() {
        return publicField + " " + protectedField;  // this object's inherited members
        // packageField would not compile: different package
    }
}
```

Listing 7.24 `two/Gamma.java`: another package, no relationship; contains `main`.

```
package two;

import one.Alpha;
import one.Beta;

public class Gamma {  // other package, not a subclass
    public static void main(String[] args) {
        Alpha a = new Alpha();
        System.out.println("Alpha    sees: " + a.allFour());
        System.out.println("Beta     sees: " + Beta.look(a));
        System.out.println("AlphaSub sees: " + new AlphaSub().look());
        System.out.println("Gamma    sees: " + a.publicField);
    }
}
```

Compile all four from the `java` folder with `javac -d out one/*.java two/*.java` and run with `java -cp out two.Gamma`:

```
Alpha    sees: public protected package private
Beta     sees: public protected package
AlphaSub sees: public protected
Gamma    sees: public
```

Each line is one column of **Table 7.5**, read top to bottom. To see the "N" entries, add these three lines to `Gamma.main` after its last line:

```
        System.out.println(a.protectedField);
        System.out.println(a.packageField);
        System.out.println(a.privateField);
```

and the compiler replies with three different reasons:

```
two/Gamma.java:13: error: protectedField has protected access in Alpha
        System.out.println(a.protectedField);
                           ^
two/Gamma.java:14: error: packageField is not public in Alpha; cannot be accessed from
outside package
        System.out.println(a.packageField);
                           ^
two/Gamma.java:15: error: privateField has private access in Alpha
        System.out.println(a.privateField);
                           ^
3 errors
```

The `protected` column has one subtlety that surprises even experienced programmers. `AlphaSub` may use `protectedField` **of its own object** (it wrote `protectedField`, meaning `this.protectedField`). It may *not* reach into an arbitrary `Alpha` object that is not an `AlphaSub`. Add this method as the first method of `AlphaSub`:

```
    public String peek(Alpha other) {
        return other.protectedField;
    }
```

and `javac` says:

```
two/AlphaSub.java:7: error: protectedField has protected access in Alpha
        return other.protectedField;
                     ^
1 error
```

The rule [3]: outside its package, a protected member may be accessed by a subclass only through references whose type is that subclass (or one of its own subclasses). The idea is that `protected` exists so that a class can share details with *its descendants*, not so that any descendant can meddle with every object of the family.

7.13.2 Choosing access levels

If other programmers use your class, you want to make sure that errors from misuse cannot happen. Access levels are the tool. Two rules of thumb cover almost everything [6], [13]:

- **Use the most restrictive access level that makes sense for a particular member. Use `private` unless you have a good reason not to.** Every member you expose is a promise you must keep forever, because somebody's code may come to depend on it.
- **Avoid public fields, except for constants** (`public static final`). A public field ties you to one representation. If `WashingMachine` exposed `public double loadKg`, you could never add a check that the load fits in the drum, because anybody could write `m.loadKg = 99`.

`protected` deserves a special caution. A protected member is part of the class's promise to *every subclass anybody will ever write*, in any package; it is nearly as public as `public`. Use it for things subclasses genuinely need, such as `comparisons` in **Listing 7.3** or `id` in **Listing 7.9**, and prefer `private` fields with `protected` methods when you can.

7.13.3 Python: conventions instead of rules

Python has no access modifiers. Everything is, technically, public. What Python has instead is a set of naming conventions, described in its style guide [14], that every Python programmer reads the same way:

- `name` — public: part of the class's promise.
- `_name` (one leading underscore) — internal: "please do not use this from outside; subclasses are welcome". The interpreter does not enforce it; tools and readers do. This is the everyday equivalent of Java's `protected` or package access, and it is what this chapter's Python listings use for fields.
- `__name` (two leading underscores, at most one trailing) — *name mangling*: inside a class `Alpha`, Python silently renames `__name` to `_Alpha__name`. Its purpose is not secrecy but *avoiding clashes*: a subclass that happens to use the same name gets its own, differently mangled attribute and cannot overwrite the parent's by accident.

Listing 7.25 shows all three.

Listing 7.25 `access_demo.py`: Python's three naming conventions, and what name mangling really does.

```python
"""access_demo.py: Python's access conventions: public, _protected, __private."""


class Alpha:
    def __init__(self):
        self.public_field = "public"
        self._protected_field = "protected"  # convention: internal, subclasses welcome
        self.__private_field = "private"     # name-mangled to _Alpha__private_field

    def all_three(self):  # inside Alpha every name works as written
        return f"{self.public_field} {self._protected_field} {self.__private_field}"


class AlphaSub(Alpha):
    def look(self):
        text = f"{self.public_field} {self._protected_field}"
        try:
            text += " " + self.__private_field  # becomes self._AlphaSub__private_field
        except AttributeError as e:
            text += f"  (AttributeError: {e})"
        return text


def main():
    a = Alpha()
    print("Alpha    sees:", a.all_three())
    print("AlphaSub sees:", AlphaSub().look())
    print("outsider sees:", a.public_field, a._protected_field, "(allowed, but impolite)")
    print("attribute names:", list(vars(a)))
    print("mangled name works:", a._Alpha__private_field, "(never do this)")


if __name__ == "__main__":
    main()
```

```
Alpha    sees: public protected private
AlphaSub sees: public protected  (AttributeError: 'AlphaSub' object has no attribute
'_AlphaSub__private_field')
outsider sees: public protected (allowed, but impolite)
attribute names: ['public_field', '_protected_field', '_Alpha__private_field']
mangled name works: private (never do this)
```

Inside `Alpha`, `self.__private_field` works as written. Inside `AlphaSub`, the very same text is mangled with the *subclass's* name to `self._AlphaSub__private_field`, which does not exist — hence the `AttributeError`. An outsider can read `_protected_field`, and the program does, with a note about manners. `vars(a)` lists the object's attributes and reveals the mangled name `_Alpha__private_field`, and the last line shows that anybody who knows the mangled name can use it. Python's philosophy is often summarized as "we are all consenting adults here": the language marks the boundary clearly and trusts you not to cross it. **Table 7.6** puts the two languages side by side.

Table 7.6 Access control in Java and its conventional counterparts in Python.

Intent	Java	Python
Anyone may use it	`public`	`name`
For subclasses (and, in Java, the package)	`protected`	`_name` (convention)
For this package / module only	no modifier	`_name` at module level; excluded from `from m import *`
For this class only	`private` (enforced by the compiler)	`__name` (mangled, not enforced)
Read-only value with computed access	`private` field + public getter	`@property` without a setter

7.14 Encapsulation: Objects That Keep Their Promises

Encapsulation means bundling data with the methods that operate on it, and hiding the data so that only those methods can change it. Its purpose is not secrecy for its own sake. Its purpose is that the class can make **promises** about its data — called **invariants** — and keep them, because no other code can break them. The idea goes back to David Parnas's 1972 paper on *information hiding*, which argued that each module should hide a design decision that is likely to change, so that when it changes, only that module must be rewritten [15]; Bertrand Meyer later made class invariants the center of his method of *design by contract* [16].

A washing machine has a capacity, fixed when it is built, and a current load. The invariant is

$$0 \le \ell \le c \tag{7.6}$$

where ℓ is the load in kilograms and c the capacity. If any code could assign to the load directly, Equation (7.6) would be a wish. With encapsulation it is a guarantee, as **Listing 7.26** shows.

Listing 7.26 `EncapsulationDemo.java`: a `WashingMachine` whose private fields can change only through methods that preserve the invariant $0 \le \ell \le c$.

```
import java.util.Locale;

public class EncapsulationDemo {
    public static void main(String[] args) {
        WashingMachine m = new WashingMachine("W-03", 8.0);
        System.out.println(m);
        m.addLoad(5.5);
        System.out.println(m);
        try {
            m.addLoad(3.0);  // would make 8.5 kg in an 8 kg drum
        } catch (IllegalArgumentException e) {
            System.out.println("Refused: " + e.getMessage());
        }
        System.out.println(m);
        m.unload();
        System.out.println(m);
        // m.loadKg = 99;  // does not compile: loadKg has private access
    }
}

class WashingMachine {
    private final String id;
    private final double capacityKg;  // fixed when the machine is built
    private double loadKg = 0.0;      // invariant: 0 <= loadKg <= capacityKg

    WashingMachine(String id, double capacityKg) {
        if (capacityKg <= 0) {
            throw new IllegalArgumentException("capacity must be positive");
        }
        this.id = id;
        this.capacityKg = capacityKg;
    }

    public double getLoadKg() {
        return loadKg;
    }

    public double getCapacityKg() {
        return capacityKg;
    }

    public double freeKg() {
        return capacityKg - loadKg;
    }
```

Listing 7.26 (Continued) *EncapsulationDemo.java: a WashingMachine whose private fields can change only through methods that preserve the invariant 0 \le \ell \le c.*

```
    public void addLoad(double kg) {
        if (kg <= 0) {
            throw new IllegalArgumentException("load must be positive");
        }
        if (kg > freeKg()) {
            throw new IllegalArgumentException(
                    String.format(Locale.US, "%.1f kg does not fit, only %.1f kg free", kg,
freeKg()));
        }
        loadKg += kg;
    }

    public void unload() {
        loadKg = 0.0;
    }

    @Override
    public String toString() {
        return String.format(Locale.US, "%s: %.1f / %.1f kg", id, loadKg, capacityKg);
    }
}
```

```
W-03: 0.0 / 8.0 kg
W-03: 5.5 / 8.0 kg
Refused: 3.0 kg does not fit, only 2.5 kg free
W-03: 5.5 / 8.0 kg
W-03: 0.0 / 8.0 kg
```

Every field is `private`; `capacityKg` is also `final`, so even the class itself cannot change it after construction. The constructor refuses a nonsensical capacity. The only method that increases the load, `addLoad`, refuses a non-positive amount and refuses an amount that does not fit, throwing `IllegalArgumentException` (Chapter 5) with a message that says why. So, whatever sequence of calls the rest of the program makes, Equation (7.6) holds after each one. The attempt `m.addLoad(3.0)` with 5.5 kg already inside is refused, and the load stays at 5.5 kg. If you remove the `//` in front of `m.loadKg = 99;` in `main`, the compiler protects the invariant for you:

```
EncapsulationDemo.java:17: error: loadKg has private access in WashingMachine
        m.loadKg = 99;
         ^
1 error
```

There is a second benefit, less visible but just as important. Because nobody outside can see `loadKg`, we are free to change how the load is stored — in grams as a `long`, say, to avoid rounding — and no other class has to change. That is Parnas's point exactly.

7.14.1 Python properties

In Python, the convention `_load_kg` tells outsiders to keep away, and methods such as `add_load` enforce the invariant just as in Java. Python adds a tool that Java lacks: the **property**. A property is a method that is *called like a field*. Its purpose is to let a class start with a plain attribute and, later, put checks behind it without changing a single line of the code that uses it. **Listing 7.27** shows both a read-only property and a property with a validating setter.

Listing 7.27 `encapsulation_demo.py`: the same machine in Python; `capacity_kg` is a read-only property and `load_kg` a property whose setter enforces the invariant.

```
"""encapsulation_demo.py: a washing machine that protects its own invariant."""


class WashingMachine:
    def __init__(self, machine_id, capacity_kg):
        if capacity_kg <= 0:
            raise ValueError("capacity must be positive")
        self._id = machine_id
        self._capacity_kg = capacity_kg  # fixed when the machine is built
        self._load_kg = 0.0              # invariant: 0 <= load <= capacity

    @property
    def capacity_kg(self):  # read-only: there is no setter
        return self._capacity_kg

    @property
    def load_kg(self):
        return self._load_kg

    @load_kg.setter
    def load_kg(self, kg):  # assignment goes through the same checks
        if kg < 0 or kg > self._capacity_kg:
            raise ValueError(f"{kg:.1f} kg is outside 0..{self._capacity_kg:.1f} kg")
        self._load_kg = kg

    def free_kg(self):
        return self._capacity_kg - self._load_kg

    def add_load(self, kg):
        if kg <= 0:
            raise ValueError("load must be positive")
        if kg > self.free_kg():
            raise ValueError(f"{kg:.1f} kg does not fit, only {self.free_kg():.1f} kg
free")
        self._load_kg += kg

    def unload(self):
        self._load_kg = 0.0

    def __str__(self):
        return f"{self._id}: {self._load_kg:.1f} / {self._capacity_kg:.1f} kg"
```

Listing 7.27 (Continued) *encapsulation_demo.py: the same machine in Python; capacity_kg is a read-only property and load_kg a property whose setter enforces the invariant.*

```
def main():
    m = WashingMachine("W-03", 8.0)
    print(m)
    m.add_load(5.5)
    print(m)
    try:
        m.add_load(3.0)  # would make 8.5 kg in an 8 kg drum
    except ValueError as e:
        print("Refused:", e)
    print(m)
    try:
        m.load_kg = 99  # looks like a field, but runs the setter
    except ValueError as e:
        print("Refused:", e)
    try:
        m.capacity_kg = 20
    except AttributeError as e:
        print("AttributeError:", e)
    m.unload()
    print(m)


if __name__ == "__main__":
    main()
```

```
W-03: 0.0 / 8.0 kg
W-03: 5.5 / 8.0 kg
Refused: 3.0 kg does not fit, only 2.5 kg free
W-03: 5.5 / 8.0 kg
Refused: 99.0 kg is outside 0..8.0 kg
AttributeError: property 'capacity_kg' of 'WashingMachine' object has no setter
W-03: 0.0 / 8.0 kg
```

The decorator `@property` above `def capacity_kg(self)` turns the method into a readable attribute: callers write `m.capacity_kg`, without parentheses. Because no setter is defined, `m.capacity_kg = 20` fails with `AttributeError`. For `load_kg`, the second decorator `@load_kg.setter` defines what happens on assignment: `m.load_kg = 99` *looks* like a field assignment but runs the setter, which checks Equation (7.6) and raises `ValueError`. This is why Python programmers rarely write `get_` and `set_` methods: they start with plain attributes and switch to properties only when a rule appears, and the callers never notice.

7.15 Profitina Laundry: the Service Hierarchy and Polymorphic Pricing

We can now rebuild the laundry's pricing so that adding a new kind of service never again means editing a `switch`. The requirements, carried over from Chapter 3 and extended:

- **Wash & Fold** (`WF`): Rp7,000 per kilogram, minimum 3 kg, ready in 48 hours.
- **Iron Only** (`IR`): Rp5,000 per kilogram, minimum 2 kg, ready in 24 hours.
- **Wash, Dry & Iron** (`WDI`): Rp10,000 per kilogram, minimum 3 kg, ready in 72 hours.
- **Dry Clean** (`DC`): priced *per piece* — shirt Rp15,000, trousers Rp18,000, kebaya Rp40,000, suit Rp45,000 — ready in 72 hours.
- **Express**: any service may be ordered express, at a 50% surcharge, ready in a quarter of the usual time but never less than 6 hours.
- **Members** get 10% off the order.

The weight-based services charge for a *chargeable weight* $\hat{w}$: the actual weight w, raised to the minimum $w_{\min}$ if it is smaller, then rounded up to the next half kilogram:

$$\hat{w} = \frac{\lceil 2\max(w, w_{\min}) \rceil}{2} \tag{7.7}$$

and the price is the chargeable weight times the rate r, in whole rupiah:

$$P_{\text{weight}} = \text{round}(r \cdot \hat{w}) \tag{7.8}$$

For example, 4.2 kg of Wash & Fold: $\max(4.2,3) = 4.2$, $2 \times 4.2 = 8.4$, rounded up to 9, halved to $\hat{w} = 4.5$ kg, so $P = 7000 \times 4.5 = 31{,}500$. An express version of any service with base price P costs, with integer division,

$$P_{\text{express}} = \left\lfloor \frac{3P}{2} \right\rfloor \tag{7.9}$$

and an order of services $s_1, \ldots, s_n$ has subtotal and member total

$$T = \sum_{i=1}^{n} P(s_i), \qquad T_{\text{member}} = T - \left\lfloor \frac{T}{10} \right\rfloor \tag{7.10}$$

The sum in Equation (7.10) is where polymorphism earns its keep: $P(s_i)$ is computed by a different formula for each kind of service, yet the code that adds them up is one loop that never asks which kind it has.

Figure 7.11 is the design.

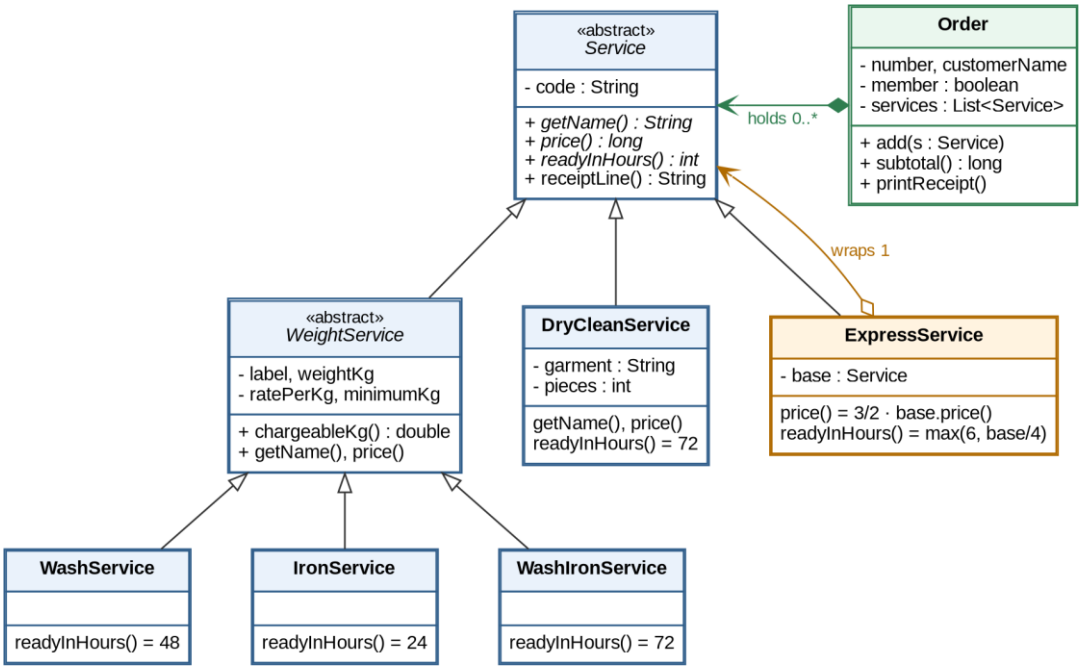


Figure 7.11 UML class diagram of the Profitina Laundry price engine. `Service` and `WeightService` are abstract; five concrete services extend them; `ExpressService` both *is* a `Service` and *wraps* one (hollow diamond); an `Order` holds any number of services (filled diamond).

- `Service` (abstract) holds what every service has: a code. It declares three abstract methods — `getName()`, `price()`, `readyInHours()` — and provides one template method, `receiptLine()`, that formats a receipt line from them.
- `WeightService` (abstract) holds everything the per-kilogram services share: the label, weight, rate and minimum, the chargeable-weight rule of Equation (7.7), and the price rule of Equation (7.8). It leaves only `readyInHours()` abstract.
- `WashService`, `IronService`, `WashIronService` are now almost empty: a constructor that passes the right numbers to `super(...)`, and a turnaround time. The `switch` of Chapter 3 has become three one-line overrides.
- `DryCleanService` extends `Service` directly, because its price depends on pieces, not weight.
- `ExpressService` extends `Service` and holds a `Service` called `base`. Its `price()` applies Equation (7.9) to `base.price()`; its turnaround is a quarter of the base's. Because it accepts *any* `Service`, express works for dry cleaning, for services added next year, and even for another express (a question for Exercise 7.9).
- `Order` holds a `List<Service>` and computes the subtotal, discount and total of Equation (7.10) with loops over that list.

Listing 7.28 is the complete program.

Listing 7.28 `LaundryPricing.java`: the Profitina Laundry service hierarchy, the express decorator, and an order that prices everything polymorphically.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

public class LaundryPricing {
    public static void main(String[] args) {
        Order order = new Order("ORD-0712", "Siti Aminah", true);
        order.add(new WashService(4.2));
        order.add(new IronService(1.3));
        order.add(new DryCleanService("suit", 1));
        order.add(new DryCleanService("shirt", 3));
        order.add(new ExpressService(new WashIronService(2.6)));
        order.add(new ExpressService(new DryCleanService("kebaya", 1)));
        order.printReceipt();
    }
}

abstract class Service {
    private final String code;

    protected Service(String code) {
        this.code = code;
    }

    String getCode() {
        return code;
    }

    abstract String getName();       // what the customer sees on the receipt
    abstract long price();           // in whole rupiah
    abstract int readyInHours();     // promised turnaround

    // A template method: fixed layout, variable parts supplied by subclasses.
    String receiptLine() {
        return String.format(Locale.US, "%-7s %-32s %10s %4d h",
                code, getName(), rupiah(price()), readyInHours());
    }

    static String rupiah(long amount) {
        return String.format(Locale.US, "Rp%,d", amount);
    }
}
```

Listing 7.28 (Continued) *LaundryPricing.java: the Profitina Laundry service hierarchy, the express decorator, and an order that prices everything polymorphically.*

```
abstract class WeightService extends Service {
    private final String label;
    private final double weightKg;
    private final long ratePerKg;
    private final double minimumKg;

    protected WeightService(String code, String label, double weightKg,
                            long ratePerKg, double minimumKg) {
        super(code);
        if (weightKg <= 0) {
            throw new IllegalArgumentException("weight must be positive");
        }
        this.label = label;
        this.weightKg = weightKg;
        this.ratePerKg = ratePerKg;
        this.minimumKg = minimumKg;
    }

    // At least the minimum, rounded up to the next 0.5 kg.
    double chargeableKg() {
        return Math.ceil(Math.max(weightKg, minimumKg) * 2) / 2;
    }

    @Override
    String getName() {
        return String.format(Locale.US, "%s %.1f kg", label, weightKg);
    }

    @Override
    long price() {
        return Math.round(chargeableKg() * ratePerKg);
    }
}

class WashService extends WeightService {
    WashService(double weightKg) {
        super("WF", "Wash & Fold", weightKg, 7000, 3.0);
    }

    @Override
    int readyInHours() {
        return 48;
    }
}
```

Listing 7.28 (Continued) *LaundryPricing.java: the Profitina Laundry service hierarchy, the express decorator, and an order that prices everything polymorphically.*

```
class IronService extends WeightService {
    IronService(double weightKg) {
        super("IR", "Iron Only", weightKg, 5000, 2.0);
    }

    @Override
    int readyInHours() {
        return 24;
    }
}

class WashIronService extends WeightService {
    WashIronService(double weightKg) {
        super("WDI", "Wash, Dry & Iron", weightKg, 10000, 3.0);
    }

    @Override
    int readyInHours() {
        return 72;
    }
}

class DryCleanService extends Service {
    private final String garment;
    private final int pieces;

    DryCleanService(String garment, int pieces) {
        super("DC");
        if (pieces <= 0) {
            throw new IllegalArgumentException("pieces must be positive");
        }
        this.garment = garment;
        this.pieces = pieces;
    }

    private long pricePerPiece() {
        return switch (garment) {
            case "shirt" -> 15000;
            case "trousers" -> 18000;
            case "kebaya" -> 40000;
            case "suit" -> 45000;
            default -> throw new IllegalArgumentException("no dry-clean price for " +
garment);
        };
    }
```

Listing 7.28 (Continued) *LaundryPricing.java: the Profitina Laundry service hierarchy, the express decorator, and an order that prices everything polymorphically.*

```
    @Override
    String getName() {
        return "Dry Clean " + pieces + " x " + garment;
    }

    @Override
    long price() {
        return pieces * pricePerPiece();
    }

    @Override
    int readyInHours() {
        return 72;
    }
}

// Express IS a Service (so an Order accepts it) and HAS a Service (the one it speeds up).
class ExpressService extends Service {
    private final Service base;

    ExpressService(Service base) {
        super("EX-" + base.getCode());
        this.base = base;
    }

    @Override
    String getName() {
        return "Express " + base.getName();
    }

    @Override
    long price() {
        return base.price() * 3 / 2;  // 50% surcharge
    }

    @Override
    int readyInHours() {
        return Math.max(6, base.readyInHours() / 4);
    }
}
```

Listing 7.28 (Continued) *LaundryPricing.java: the Profitina Laundry service hierarchy, the express decorator, and an order that prices everything polymorphically.*

```
class Order {
    private final String number;
    private final String customerName;
    private final boolean member;
    private final List<Service> services = new ArrayList<>();

    Order(String number, String customerName, boolean member) {
        this.number = number;
        this.customerName = customerName;
        this.member = member;
    }

    void add(Service s) {
        services.add(s);
    }

    long subtotal() {
        long sum = 0;
        for (Service s : services) {
            sum += s.price();  // dynamic dispatch picks the right price()
        }
        return sum;
    }

    long discount() {
        return member ? subtotal() * 10 / 100 : 0;
    }

    int readyInHours() {
        int latest = 0;
        for (Service s : services) {
            latest = Math.max(latest, s.readyInHours());
        }
        return latest;
    }
```

Listing 7.28 (Continued) *LaundryPricing.java: the Profitina Laundry service hierarchy, the express decorator, and an order that prices everything polymorphically.*

```
    void printReceipt() {
        System.out.println("PROFITINA LAUNDRY  order " + number + "  customer " +
customerName
                + (member ? " (member)" : ""));
        for (Service s : services) {
            System.out.println(s.receiptLine());
        }
        System.out.printf("%-40s %10s%n", "Subtotal", Service.rupiah(subtotal()));
        if (member) {
            System.out.printf("%-40s %10s%n", "Member discount 10%", "-" +
Service.rupiah(discount()));
        }
        System.out.printf("%-40s %10s%n", "TOTAL", Service.rupiah(subtotal() -
discount()));
        System.out.println("Everything ready in " + readyInHours() + " h");
    }
}
```

```
PROFITINA LAUNDRY  order ORD-0712  customer Siti Aminah (member)
WF      Wash & Fold 4.2 kg                Rp31,500   48 h
IR      Iron Only 1.3 kg                  Rp10,000   24 h
DC      Dry Clean 1 x suit                Rp45,000   72 h
DC      Dry Clean 3 x shirt               Rp45,000   72 h
EX-WDI  Express Wash, Dry & Iron 2.6 kg   Rp45,000   18 h
EX-DC   Express Dry Clean 1 x kebaya      Rp60,000   18 h
Subtotal                                 Rp236,500
Member discount 10%                      -Rp23,650
TOTAL                                    Rp212,850
Everything ready in 72 h
```

Check each line against the formulas. Wash & Fold 4.2 kg is Rp31,500, as computed above. Iron Only 1.3 kg is raised to the 2 kg minimum: $5000 \times 2 = 10{,}000$. One suit is Rp45,000; three shirts are $3 \times 15{,}000 = 45{,}000$. Wash, Dry & Iron 2.6 kg is raised to 3 kg, $10{,}000 \times 3 = 30{,}000$, and express makes it $\lfloor 3 \times 30{,}000/2 \rfloor = 45{,}000$, ready in $\max(6, 72/4) = 18$ hours. The express kebaya is $\lfloor 3 \times 40{,}000/2 \rfloor = 60{,}000$. The subtotal is $31{,}500 + 10{,}000 + 45{,}000 + 45{,}000 + 45{,}000 + 60{,}000 = 236{,}500$; the member discount is $\lfloor 236{,}500/10 \rfloor = 23{,}650$; the total is 212,850. The order is ready when its slowest item is: 72 hours.

Now look at what the code does *not* contain. `Order.subtotal()` is four lines and mentions no subclass. `Service.receiptLine()` formats every line and mentions no subclass. There is no `switch` on a code and no `if (express)` anywhere. The one remaining `switch`, inside `DryCleanService`, chooses a *number* from a price list, which is data, not behavior. When the laundry adds shoe cleaning, we will write one new class and change **nothing** that exists — as Exercise 7.9 and its solution demonstrate. That property, being open to extension while closed to modification, is the practical payoff of this whole chapter. And if we forget part of the contract, the compiler reminds us: delete `readyInHours()` from `IronService` and `javac` refuses the class, naming the missing method:

```
LaundryPricing.java:89: error: IronService is not abstract and does not override abstract
method readyInHours() in Service
class IronService extends WeightService {
^
1 error
```

The design also uses every access level with intent. The fields are `private`, so no subclass can corrupt another class's data; subclasses reach their parent's data only through methods such as `getCode()`. The classes are package-private, because only this program uses them. The `rupiah` helper is `static` and passes `Locale.US`, so that the thousands separator is a comma on every computer, matching the Python output exactly.

Listing 7.29 is the Python version, and its output is identical.

Listing 7.29 `laundry_pricing.py`: the same price engine in Python.

```
"""laundry_pricing.py: the Profitina Laundry service hierarchy and polymorphic pricing."""
import math
from abc import ABC, abstractmethod


def rupiah(amount):
    return f"Rp{amount:,}"


class Service(ABC):
    def __init__(self, code):
        self._code = code

    @property
    def code(self):
        return self._code

    @abstractmethod
    def name(self): ...           # what the customer sees on the receipt

    @abstractmethod
    def price(self): ...          # in whole rupiah

    @abstractmethod
    def ready_in_hours(self): ...  # promised turnaround

    # A template method: fixed layout, variable parts supplied by subclasses.
    def receipt_line(self):
        return (f"{self._code:<7} {self.name():<32} {rupiah(self.price()):>10} "
                f"{self.ready_in_hours():4d} h")


class WeightService(Service, ABC):
    def __init__(self, code, label, weight_kg, rate_per_kg, minimum_kg):
        super().__init__(code)
        if weight_kg <= 0:
            raise ValueError("weight must be positive")
        self._label = label
        self._weight_kg = weight_kg
        self._rate_per_kg = rate_per_kg
        self._minimum_kg = minimum_kg

    def chargeable_kg(self):
        # At least the minimum, rounded up to the next 0.5 kg.
        return math.ceil(max(self._weight_kg, self._minimum_kg) * 2) / 2
```

Listing 7.29 (Continued) *laundry_pricing.py: the same price engine in Python.*

```
    def name(self):
        return f"{self._label} {self._weight_kg:.1f} kg"

    def price(self):
        return round(self.chargeable_kg() * self._rate_per_kg)


class WashService(WeightService):
    def __init__(self, weight_kg):
        super().__init__("WF", "Wash & Fold", weight_kg, 7000, 3.0)

    def ready_in_hours(self):
        return 48


class IronService(WeightService):
    def __init__(self, weight_kg):
        super().__init__("IR", "Iron Only", weight_kg, 5000, 2.0)

    def ready_in_hours(self):
        return 24


class WashIronService(WeightService):
    def __init__(self, weight_kg):
        super().__init__("WDI", "Wash, Dry & Iron", weight_kg, 10000, 3.0)

    def ready_in_hours(self):
        return 72


class DryCleanService(Service):
    PRICE_PER_PIECE = {"shirt": 15000, "trousers": 18000, "kebaya": 40000, "suit": 45000}

    def __init__(self, garment, pieces):
        super().__init__("DC")
        if pieces <= 0:
            raise ValueError("pieces must be positive")
        if garment not in self.PRICE_PER_PIECE:
            raise ValueError(f"no dry-clean price for {garment}")
        self._garment = garment
        self._pieces = pieces

    def name(self):
        return f"Dry Clean {self._pieces} x {self._garment}"
```

Listing 7.29 (Continued) *laundry_pricing.py: the same price engine in Python.*

```
    def price(self):
        return self._pieces * self.PRICE_PER_PIECE[self._garment]

    def ready_in_hours(self):
        return 72


# Express IS a Service (so an Order accepts it) and HAS a Service (the one it speeds up).
class ExpressService(Service):
    def __init__(self, base):
        super().__init__("EX-" + base.code)
        self._base = base

    def name(self):
        return "Express " + self._base.name()

    def price(self):
        return self._base.price() * 3 // 2  # 50% surcharge

    def ready_in_hours(self):
        return max(6, self._base.ready_in_hours() // 4)


class Order:
    def __init__(self, number, customer_name, member):
        self._number = number
        self._customer_name = customer_name
        self._member = member
        self._services = []

    def add(self, service):
        self._services.append(service)

    def subtotal(self):
        return sum(s.price() for s in self._services)  # each s picks its own price()

    def discount(self):
        return self.subtotal() * 10 // 100 if self._member else 0

    def ready_in_hours(self):
        return max((s.ready_in_hours() for s in self._services), default=0)
```

Listing 7.29 (Continued) *laundry_pricing.py: the same price engine in Python.*

```
    def print_receipt(self):
        member = " (member)" if self._member else ""
        print(f"PROFITINA LAUNDRY  order {self._number}  customer
{self._customer_name}{member}")
        for s in self._services:
            print(s.receipt_line())
        print(f"{'Subtotal':<40} {rupiah(self.subtotal()):>10}")
        if self._member:
            print(f"{'Member discount 10%':<40} {'-' + rupiah(self.discount()):>10}")
        print(f"{'TOTAL':<40} {rupiah(self.subtotal() - self.discount()):>10}")
        print(f"Everything ready in {self.ready_in_hours()} h")


def main():
    order = Order("ORD-0712", "Siti Aminah", True)
    order.add(WashService(4.2))
    order.add(IronService(1.3))
    order.add(DryCleanService("suit", 1))
    order.add(DryCleanService("shirt", 3))
    order.add(ExpressService(WashIronService(2.6)))
    order.add(ExpressService(DryCleanService("kebaya", 1)))
    order.print_receipt()


if __name__ == "__main__":
    main()
```

```
PROFITINA LAUNDRY  order ORD-0712  customer Siti Aminah (member)
WF      Wash & Fold 4.2 kg                Rp31,500   48 h
IR      Iron Only 1.3 kg                  Rp10,000   24 h
DC      Dry Clean 1 x suit                Rp45,000   72 h
DC      Dry Clean 3 x shirt               Rp45,000   72 h
EX-WDI  Express Wash, Dry & Iron 2.6 kg   Rp45,000   18 h
EX-DC   Express Dry Clean 1 x kebaya      Rp60,000   18 h
Subtotal                                 Rp236,500
Member discount 10%                      -Rp23,650
TOTAL                                    Rp212,850
Everything ready in 72 h
```

The structure is the same class for class. `WeightService(Service, ABC)` lists `ABC` again to make its abstractness explicit. `//` is Python's integer division, matching Java's `/` on `long` values. Python's `round` and Java's `Math.round` treat exact halves differently (Python rounds them to the nearest even number), but with the prices of this laundry the product $r \cdot \widehat{w}$ is always a whole number, so the two agree. `f"{amount:,}"` inserts commas between thousands regardless of the computer's regional settings, so no locale is needed. And the heart of the program, `sum(s.price() for s in self._services)`, is the sum of Equation (7.10) written almost exactly as mathematics writes it.

7.16 Profitina in Practice

The laundry is our classroom; Profitina's real product uses the same ideas at a larger scale. Profitina is an AI visibility platform: it scans a brand's website and measures how ready the site is to be quoted by AI answer engines — the discipline called *generative engine optimization* (GEO). A scan examines many independent **signals**, such as whether the content can be extracted cleanly, whether the page carries schema.org structured data, and whether `robots.txt` lets answer engines in.

The shape of that scanner is the shape of **Figure 7.11**. One abstraction — a signal — promises a few operations: measure a page, give a score, explain the score in words a brand owner can

understand. Many concrete signal classes implement that promise, each tested on its own. The code that runs a scan is one loop over a list of signals, just as `Order.subtotal()` is one loop over services, and it never asks which signal it holds. A new signal is one new class, not an edit to the loop; a behavior shared by several signals lives in a helper object that they compose, for the reasons of Section 7.11.

Encapsulation carries weight there too. Profitina complies with Indonesia's personal data protection law (UU PDP No. 27/2022): account data is hashed with Argon2id, consent is explicit, and a user can export or purge their data. Such code keeps personal data behind a small set of methods that enforce the rules, in the spirit of **Listing 7.26**. And, as everywhere in Profitina, the scores are advice: the platform explains what it measured; the user decides what to do.

7.17 Common Mistakes, and How to Avoid Them

1. Inheriting for code reuse when the is-a sentence is false. `class Order extends ArrayList<Service>` saves a few lines and gives every order `clear()`, `remove()` and forty other methods nobody wanted. *Avoid it:* say the is-a sentence aloud; if it fails, hold the list in a private field (Section 7.11).

2. Overloading when you meant to override. In Java, `public boolean equals(Customer other)` does *not* override `equals(Object)`; it adds an overload that `HashSet` never calls. *Avoid it:* always write `@Override`. With it, the mistake is a compile error:

```
BadEquals.java:9: error: method does not override or implement a method from a supertype
    @Override
    ^
1 error
```

A misspelled name (`Start` for `start`) is caught the same way.

3. Forgetting `super().__init__()` in Python. The superclass's fields are never created, and the error appears much later as an `AttributeError` (third block of **Listing 7.6**). *Avoid it:* make `super().__init__(...)` the first line of every subclass `__init__`.

4. Redeclaring a superclass field in a Java subclass. The object then holds two fields with the same name, and which one you get depends on the static type (**Listing 7.9**). *Avoid it:* never redeclare; if a subclass needs a different value, pass it to the superclass constructor or override a getter.

5. Expecting static methods or fields to be polymorphic. `m.kind()` calls `Machine.kind()` even when `m` holds a `Washer`. *Avoid it:* call static methods through the class name, `Washer.kind()`, so the code says what it does.

6. Downcasting without checking. `(Dryer) floor[0]` compiles and throws `ClassCastException` at run time. *Avoid it:* test with `if (x instanceof Dryer d)`; better still, ask whether the method should be declared in the superclass so that no cast is needed.

7. Overriding `equals` without `hashCode`. Hash-based collections then misbehave silently in Java. *Avoid it:* override both, from the same fields (Section 7.8). In Python, define `__hash__` whenever you define `__eq__` on a class whose objects go into sets or dictionary keys.

8. Making fields `protected` "so subclasses can use them". Every subclass, in any package, can then break the invariant. *Avoid it:* keep fields `private` and give subclasses `protected` methods that preserve the rules.

9. Reading `super()` as "my parent" in Python multiple inheritance. With a diamond, `super()` goes to the next class in the MRO, which may be a sibling (**Listing 7.20**). *Avoid it:* print `Class.__mro__` when in doubt, and design mixins to call `super()` cooperatively.

10. Calling an overridable method from a constructor. Because of dynamic dispatch, a superclass constructor that calls `describe()` runs the *subclass's* `describe()` — before the subclass's constructor has set the subclass's fields. *Avoid it:* constructors should call only `private` or `final` methods [6].

7.18 Chapter Summary

- **Inheritance** removes duplicated code: the common part of several classes moves into a **superclass**, and each **subclass** declares only what differs (Java `class C extends B`, Python `class C(B)`). The design test is the sentence "a C *is a* B".
- A subclass **overrides** an inherited method by redefining it (Java: same name and parameter types, checked by `@Override`; Python: same name). **Overloading** — same name, different parameters — is resolved at compile time; Python replaces it with default and keyword arguments.
- **Constructors chain upward**: the superclass part is built first. Java inserts `super()` if you write no `super(...)`; Python runs the superclass `__init__` only if you call `super().__init__(...)`. `super.method()` / `super().method()` reaches the overridden version.
- **Polymorphism**: one call runs different code for different objects. In Java the **static type** decides whether a call compiles and the **dynamic type** decides which override runs (**dynamic dispatch**, Equation (7.1)), implemented with one method table per class; fields, static and private methods are not dispatched. Python looks names up along the **MRO** at run time.
- Every class descends from `Object` / `object`; override `toString`/`__str__`, and `equals` together with `hashCode` (`__eq__` with `__hash__`).
- An **abstract class** cannot be instantiated; it gathers shared state and code, declares what subclasses must implement, and enables **template methods** such as `describe()` and `receiptLine()`. Prefer an abstract common superclass to inheriting from a concrete class.
- **Favor composition over inheritance**: inheritance couples a subclass to its superclass's implementation (the fragile base class). Inherit only when the **Liskov substitution principle** (Equation (7.4)) holds. The express service does both — it extends and wraps `Service`, a decorator.
- Java allows **one superclass and many interfaces**, and conflicting defaults must be resolved explicitly. Python allows **multiple inheritance**, ordered by the **C3 linearization** (Equation (7.5)), which `super()` follows.
- **Access control**: Java's `public`, `protected`, package-private and `private` (**Table 7.4**) — choose the most restrictive that works. Python's conventions are `name`, `_name` and `__name`. **Encapsulation** lets a class keep its **invariants**; `@property` puts checks behind attribute syntax.
- In the laundry, one loop now prices every kind of service, and a new service is a new class, not a new `case` label.

7.19 Exercises

Exercise 7.1 (is-a or has-a). For each pair, say whether the first should *inherit from* the second, *hold* an object of the second, or neither, and give a one-sentence reason. (a) `ExpressService`,

`Service`. (b) `Order`, `Customer`. (c) `DryCleanService`, `Service`. (d) `WashingMachine`, `Machine`. (e) `Receipt`, `Order`. (f) `Customer`, `Order`. (g) `ProfitinaLaundry`, `WashingMachine`.

Exercise 7.2 (tracing). For `TreeIntDB` in **Listing 7.3**, draw the tree after inserting 42, 7, 19, 3, 25, and reproduce the comparison count 22 by listing, for each of the seven searches the program makes, the nodes visited. Then predict the count for `LinearIntDB` if the keys had been added in the order 3, 7, 19, 25, 42, and check your answer by changing the program.

Exercise 7.3 (overloading or overriding). In the following Java code, state for each of the lines marked (1)–(5) which method runs and what is printed, and label each relationship as overloading or overriding.

```
class Bag {
    String weigh()            { return "Bag.weigh()"; }
    String weigh(double kg)   { return "Bag.weigh(double)"; }
}
class LaundryBag extends Bag {
    @Override String weigh()  { return "LaundryBag.weigh()"; }
    String weigh(int grams)   { return "LaundryBag.weigh(int)"; }
}
Bag b = new LaundryBag();
LaundryBag lb = new LaundryBag();
b.weigh();        // (1)
b.weigh(2.5);     // (2)
b.weigh(2500);    // (3)
lb.weigh(2500);   // (4)
lb.weigh(2.5);    // (5)
```

Exercise 7.4 (a hierarchy of your own). Implement the student hierarchy on the right of **Figure 7.5** in Java and in Python. `Student` and `TaughtStudent` are abstract. A semester's fee is: undergraduate, Rp300,000 per credit; postgraduate, Rp500,000 per credit; occasional, Rp250,000 registration plus Rp400,000 per credit; research, a flat Rp9,000,000. Print one line per student and the total for Ayu (undergraduate, 20 credits), Bagus (postgraduate, 12), Citra (occasional, 6) and Dewi (research).

Exercise 7.5 (predict the output). Without running it, predict every line printed by the first block of **Listing 7.9** if (a) `Washer` did *not* redeclare `label`; (b) `describe()` in `Machine` were made `private` and called from a new public method `info()` in `Machine`; (c) `kind()` were an instance method rather than `static`. Justify each answer with **Table 7.2**.

Exercise 7.6 (C3 by hand). Given

```
class A: ...
class B(A): ...
class C(A): ...
class D(B, C): ...
class E(C): ...
class F(D, E): ...
```

compute $L[F]$ with Equation (7.5), showing every step of the merge. Then check your answer with `print([k.__name__ for k in F.__mro__])`. Finally, explain why `class G(A, B)` raises `TypeError`.

Exercise 7.7 (access). In the setting of **Figure 7.10**, suppose `Alpha` also declares `protected void touch()` and `static protected int counter`. For each of the following statements, say whether it compiles, and why: (a) in `Beta`, `new Alpha().touch();` (b) in `AlphaSub`, `touch();` (c) in `AlphaSub`, `new Alpha().touch();` (d) in `AlphaSub`, `new AlphaSub().touch();` (e) in `Gamma`, `new AlphaSub().touch();`.

Exercise 7.8 (redesign). The concrete-superclass version of the database (**Listing 7.15**) makes `TreeDB` inherit from `LinearDB`. List every problem this causes, and redesign it in two ways: with an abstract superclass, and with an interface `KeyStore` plus a helper class used by composition. Which design would you choose if a third implementation, a hash table, were planned?

Exercise 7.9 (extend the laundry). Without changing `LaundryPricing.java` or `laundry_pricing.py` at all, add (a) a `ShoeCleaningService` priced at Rp25,000 per pair and ready in 72 hours, and (b) a `PromoService` that wraps any service and takes a given percentage off its price. Price an order for Budi (not a member) containing 3.4 kg of Wash & Fold, two pairs of shoes, one pair of shoes at 20% off, and an express Wash & Fold of 5.0 kg at 20% off. Does it matter whether the promotion is applied inside or outside the express? What does `new ExpressService(new ExpressService(s))` cost?

Exercise 7.10 (Python name mangling). Predict the output of `print(list(vars(Child())))` for

```
class Parent:
    def __init__(self):
        self.__secret = 1
        self._hint = 2
class Child(Parent):
    def __init__(self):
        super().__init__()
        self.__secret = 3
```

and explain why the child's assignment did not overwrite the parent's value.

Exercise 7.11 (substitution). The laundry is asked to add `FlatRateWash`, a subclass of `WashService` for which the customer pays a fixed Rp50,000 per bag "whatever the weight", and `chargeableKg()` is overridden to return 0 so that the receipt shows no weight. Explain, using Equation (7.4), which callers could be broken by this subclass, and propose a better design.

Exercise 7.12 (properties). Rewrite `Customer` from **Listing 7.12** so that `phone` is a read-only property, `name` is a property whose setter strips surrounding spaces and rejects an empty name, and the class still works in a set. Explain why `phone` in particular must not be settable once the object is in a set.

Exercise 7.13 (constructors calling overridable methods). Write a Java class `Item` whose constructor prints `describe()`, and a subclass `Towel` that overrides `describe()` to print a field `color` set in `Towel`'s constructor. Run `new Towel("white")` and explain the output with **Figure 7.2**. Do the same in Python and compare.

7.20 Solutions to the Exercises

Solution 7.1. (a) *Inherit and hold*: an express service is a service (an order must accept it) and it has the service it speeds up — the decorator of Section 7.15. (b) *Hold*: an order has a customer; an order is not a customer. (c) *Inherit*: dry cleaning is a service, and every caller of `price()` is happy with it. (d) *Inherit*, provided every `Machine` operation makes sense for a washing machine. (e) *Hold* (or neither): a receipt is produced from an order and may refer to it, but it is not an order — it cannot have services added to it. (f) *Neither directly*; at most a customer holds a list of past orders. (g) *Hold*: the business has machines. In every "hold" case, the is-a sentence sounds absurd when said aloud, which is the quickest test.

Solution 7.2. The tree has root 42; 42's left child is 7; 7's left child is 3 and right child 19; 19's right child is 25 (42 has no right child). The searches and the nodes they visit are: `search(19)`: 42, 7, 19 (3); `search(8)`: 42, 7, 19, then the empty left child of 19 (3); `findOneOf({8,9,25})`: 8 visits

42, 7, 19 (3), 9 visits 42, 7, 19 (3), 25 visits 42, 7, 19, 25 and is found (4); `findOneOf({1,2})`: 1 visits 42, 7, 3 (3), 2 visits 42, 7, 3 (3). Total $3 + 3 + 3 + 3 + 4 + 3 + 3 = 22$. For `LinearIntDB` with the keys stored in the order 3, 7, 19, 25, 42: `search(19)` costs 3, `search(8)` costs 5, then 8, 9 cost 5 each and 25 costs 4, then 1 and 2 cost 5 each, total $3 + 5 + 5 + 5 + 4 + 5 + 5 = 32$. Change the array in `main` to `{3, 7, 19, 25, 42}` to confirm. (Note that the tree built from this sorted order is a chain — every node has only a right child — so the tree loses its advantage; Chapter 9 and a data structures course explain balanced trees.)

Solution 7.3. Running the code prints:

```
LaundryBag.weigh()
Bag.weigh(double)
Bag.weigh(double)
LaundryBag.weigh(int)
Bag.weigh(double)
```

(1)`weigh()` is overridden, so the dynamic type `LaundryBag` decides: *overriding*. (2) Overload resolution uses the static type `Bag`, which has `weigh()` and `weigh(double)`; `2.5` matches `weigh(double)`, which `LaundryBag` does not override. (3) The surprise: the static type is still `Bag`, which knows nothing of `weigh(int)`; the compiler widens `2500` to `2500.0` and chooses `weigh(double)`. `LaundryBag.weigh(int)` is an *overload* added by the subclass, not an override, so dispatch never considers it. (4) With static type `LaundryBag`, both `weigh(double)` (inherited) and `weigh(int)` are candidates, and `int` matches `2500` exactly. (5) `2.5` is a `double`, so the inherited `Bag.weigh(double)` runs. Lesson: overloads are chosen by the compiler from static types; only overrides are dispatched.

Solution 7.4. **Listing 7.30** and **Listing 7.31** implement the hierarchy. `TaughtStudent` holds the credits and overrides `kind()` to append them, calling `super.kind()`; `ResearchStudent` extends `Student` directly because it has no credits.

Listing 7.30 `StudentFees.java`: the student hierarchy of **Figure 7.5** with polymorphic fees.

```
import java.util.Locale;

public class StudentFees {
    public static void main(String[] args) {
        Student[] students = {
            new UndergraduateStudent("Ayu", 20),
            new PostgraduateStudent("Bagus", 12),
            new OccasionalStudent("Citra", 6),
            new ResearchStudent("Dewi")
        };
        long total = 0;
        for (Student s : students) {
            System.out.println(s.line());
            total += s.fee();
        }
        System.out.println(String.format(Locale.US, "%-40s Rp%,d", "Total", total));
    }
}

abstract class Student {
    private final String name;

    protected Student(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    abstract long fee();  // tuition for one semester, in rupiah

    String kind() {
        return getClass().getSimpleName();
    }

    String line() {
        return String.format(Locale.US, "%-6s %-33s Rp%,d", name, kind(), fee());
    }
}

abstract class TaughtStudent extends Student {
    private final int credits;
```

Listing 7.30 (Continued) *StudentFees.java: the student hierarchy of Figure 7.5 with polymorphic fees.*

```
    protected TaughtStudent(String name, int credits) {
        super(name);
        this.credits = credits;
    }

    int getCredits() {
        return credits;
    }

    @Override
    String kind() {
        return super.kind() + " (" + credits + " credits)";
    }
}

class UndergraduateStudent extends TaughtStudent {
    UndergraduateStudent(String name, int credits) {
        super(name, credits);
    }

    @Override
    long fee() {
        return getCredits() * 300_000L;
    }
}

class PostgraduateStudent extends TaughtStudent {
    PostgraduateStudent(String name, int credits) {
        super(name, credits);
    }

    @Override
    long fee() {
        return getCredits() * 500_000L;
    }
}

class OccasionalStudent extends TaughtStudent {
    OccasionalStudent(String name, int credits) {
        super(name, credits);
    }
```

Listing 7.30 (Continued) *StudentFees.java: the student hierarchy of Figure 7.5 with polymorphic fees.*

```
    @Override
    long fee() {
        return 250_000L + getCredits() * 400_000L;  // registration plus credits
    }
}

class ResearchStudent extends Student {
    ResearchStudent(String name) {
        super(name);
    }

    @Override
    long fee() {
        return 9_000_000L;  // flat fee: no taught credits
    }
}
```

```
Ayu    UndergraduateStudent (20 credits) Rp6,000,000
Bagus  PostgraduateStudent (12 credits)  Rp6,000,000
Citra  OccasionalStudent (6 credits)     Rp2,650,000
Dewi   ResearchStudent                   Rp9,000,000
Total                                    Rp23,650,000
```

Listing 7.31 `student_fees.py`: the student hierarchy in Python.

```python
"""student_fees.py: a student hierarchy with polymorphic fees."""
from abc import ABC, abstractmethod


class Student(ABC):
    def __init__(self, name):
        self._name = name

    @abstractmethod
    def fee(self): ...  # tuition for one semester, in rupiah

    def kind(self):
        return type(self).__name__

    def line(self):
        return f"{self._name:<6} {self.kind():<33} Rp{self.fee():,}"


class TaughtStudent(Student, ABC):
    def __init__(self, name, credits):
        super().__init__(name)
        self._credits = credits

    def kind(self):
        return super().kind() + f" ({self._credits} credits)"


class UndergraduateStudent(TaughtStudent):
    def fee(self):
        return self._credits * 300_000


class PostgraduateStudent(TaughtStudent):
    def fee(self):
        return self._credits * 500_000


class OccasionalStudent(TaughtStudent):
    def fee(self):
        return 250_000 + self._credits * 400_000  # registration plus credits


class ResearchStudent(Student):
    def fee(self):
        return 9_000_000  # flat fee: no taught credits
```

Listing 7.31 (Continued) *student_fees.py: the student hierarchy in Python.*

```
def main():
    students = [UndergraduateStudent("Ayu", 20), PostgraduateStudent("Bagus", 12),
                OccasionalStudent("Citra", 6), ResearchStudent("Dewi")]
    for s in students:
        print(s.line())
    print(f"{'Total':<40} Rp{sum(s.fee() for s in students):,}")


if __name__ == "__main__":
    main()
```

```
Ayu    UndergraduateStudent (20 credits) Rp6,000,000
Bagus  PostgraduateStudent (12 credits)  Rp6,000,000
Citra  OccasionalStudent (6 credits)     Rp2,650,000
Dewi   ResearchStudent                   Rp9,000,000
Total                                    Rp23,650,000
```

Check: $20 \times 300{,}000 = 6{,}000{,}000$; $12 \times 500{,}000 = 6{,}000{,}000$; $250{,}000 + 6 \times 400{,}000 = 2{,}650{,}000$; total $6 + 6 + 2.65 + 9 = 23.65$ million rupiah. `getClass().getSimpleName()` and `type(self).__name__` return the dynamic class name, so `kind()` needs no override in the concrete classes.

Solution 7.5. (a) Without the redeclaration there is only one `label` field, so `m.label` prints `machine` and `describe()` prints `W-01 is a machine` — exactly as before, because the static type `Machine` already selected `Machine`'s field. Removing the hidden field changes nothing in the output but removes the trap. (b) A private method is not dispatched, but here there is nothing to dispatch anyway: `Washer` does not override `describe`. `m.info()` calls `Machine.describe()` and the output is unchanged; the experiment only proves that private methods can be called from inside their own class. (c) As an instance method, `kind()` becomes an override, and `m.kind()` now prints `Washer.kind()`: the dynamic type decides. The first four lines therefore read `W-01: fill water, spin at 1200 rpm`, `m.label  = machine`, `m.kind() = Washer.kind()`, `m.describe() = W-01 is a machine`.

Solution 7.6. First the parts: $L[A] = [A, o]$ (writing o for `object`); $L[B] = [B, A, o]$; $L[C] = [C, A, o]$; $L[E] = [E, C, A, o]$. For $D(B, C)$: merge $[B, A, o]$, $[C, A, o]$, $[B, C]$: take B; A is in the tail of $[C, A, o]$, so take C; then A, o. So $L[D] = [D, B, C, A, o]$. For $F(D, E)$: merge $[D, B, C, A, o]$, $[E, C, A, o]$, $[D, E]$:

1. D: not in any tail; take it. Lists: $[B, C, A, o]$, $[E, C, A, o]$, $[E]$.
2. B: not in any tail; take it. Lists: $[C, A, o]$, $[E, C, A, o]$, $[E]$.
3. C: in the tail of $[E, C, A, o]$; skip. Next head E: not in any tail; take it. Lists: $[C, A, o]$, $[C, A, o]$, $[\,]$.
4. C, then A, then o.

$L[F] = [F, D, B, E, C, A, o]$. Python agrees:

```
['F', 'D', 'B', 'E', 'C', 'A', 'object']
TypeError: Cannot create a consistent method resolution
order (MRO) for bases A, B
```

For `G(A, B)` the merge is over $[A, o]$, $[B, A, o]$, $[A, B]$. The head A is in the tail of $[B, A, o]$; the next head, B, is in the tail of $[A, B]$. No head is acceptable, so no order can list `A` before `B` (as requested) and `B` before `A` (as a subclass must be). Hence the `TypeError`.

Solution 7.7. (a) Compiles: `Beta` is in the same package, and `protected` includes package access. (b) Compiles: a subclass may use a protected member of its own object. (c) Does not compile:

outside package `one`, `AlphaSub` may use `touch()` only through a reference of type `AlphaSub`, and `new Alpha()` is not one. (d) Compiles for the same reason, since `new AlphaSub()` has type `AlphaSub`; the static field `counter` is accessible from `AlphaSub` as `counter` or `Alpha.counter`. (e) Does not compile: `Gamma` is neither in package `one` nor a subclass. To check, add `touch()` and `counter` to `Alpha`; put (a) in a new static method of `Beta`; put (b), (c), (d), `counter++` and `Alpha.counter++` in a new method at the end of `AlphaSub`; and put (e) as the last statement of `Gamma.main`. `javac` then reports exactly two errors:

```
two/AlphaSub.java:13: error: touch() has protected access in Alpha
        new Alpha().touch();         // (c)
                    ^
two/Gamma.java:13: error: touch() has protected access in Alpha
        new AlphaSub().touch();          // (e)
                       ^
2 errors
```

Solution 7.8. The problems are those of Section 7.10: every `TreeDB` carries an unused array and counter; a `TreeDB` can be passed wherever a `LinearDB` is expected, although it is not one; and `TreeDB` depends on `LinearDB`'s implementation, so a change there (for example, making `addSeveral` stop calling `addKey`) silently changes `TreeDB`. *Design 1* is **Listing 7.1**: an abstract `DB` with the two shared methods and two abstract ones. *Design 2*: an interface `KeyStore` declaring `addKey` and `search`; classes `ArrayKeyStore` and `TreeKeyStore` implementing it; and a class `KeyBatch` that holds a `KeyStore` and implements `addSeveral` and `findOneOf` by calling it. For a planned third implementation, both work: a `HashDB extends DB` in design 1, or a `HashKeyStore implements KeyStore` in design 2. Design 1 is simpler when every store needs the batch operations; design 2 is more flexible when stores must also extend some other class, since Java allows only one superclass but many interfaces.

Solution 7.9. **Listing 7.32** adds two classes and changes nothing that exists. `ShoeCleaningService` extends `Service`; `PromoService` extends `Service` and wraps one, exactly like `ExpressService`. Compile it together with the price engine, `javac LaundryPricing.java ShoePromoDemo.java`, then run `java ShoePromoDemo`.

Listing 7.32 `ShoePromoDemo.java`: shoe cleaning and promotions added to the price engine without modifying it.

```
// Compile together with LaundryPricing.java:  javac LaundryPricing.java ShoePromoDemo.java
public class ShoePromoDemo {
    public static void main(String[] args) {
        Order order = new Order("ORD-0713", "Budi Santoso", false);
        order.add(new WashService(3.4));
        order.add(new ShoeCleaningService(2));
        order.add(new PromoService(new ShoeCleaningService(1), 20));
        order.add(new ExpressService(new PromoService(new WashService(5.0), 20)));
        order.printReceipt();
    }
}

class ShoeCleaningService extends Service {
    private final int pairs;

    ShoeCleaningService(int pairs) {
        super("SH");
        if (pairs <= 0) {
            throw new IllegalArgumentException("pairs must be positive");
        }
        this.pairs = pairs;
    }

    @Override
    String getName() {
        return "Shoe Cleaning " + pairs + " pair" + (pairs == 1 ? "" : "s");
    }

    @Override
    long price() {
        return pairs * 25000L;
    }

    @Override
    int readyInHours() {
        return 72;
    }
}

// A promotion wraps any service, exactly as ExpressService does.
class PromoService extends Service {
    private final Service base;
    private final int percentOff;
```

Listing 7.32 (Continued) *ShoePromoDemo.java: shoe cleaning and promotions added to the price engine without modifying it.*

```
    PromoService(Service base, int percentOff) {
        super("P-" + base.getCode());
        if (percentOff <= 0 || percentOff >= 100) {
            throw new IllegalArgumentException("percentOff must be between 1 and 99");
        }
        this.base = base;
        this.percentOff = percentOff;
    }

    @Override
    String getName() {
        return base.getName() + " -" + percentOff + "%";
    }

    @Override
    long price() {
        return base.price() - base.price() * percentOff / 100;
    }

    @Override
    int readyInHours() {
        return base.readyInHours();
    }
}
```

```
PROFITINA LAUNDRY  order ORD-0713  customer Budi Santoso
WF      Wash & Fold 3.4 kg                Rp24,500   48 h
SH      Shoe Cleaning 2 pairs             Rp50,000   72 h
P-SH    Shoe Cleaning 1 pair -20%         Rp20,000   72 h
EX-P-WF Express Wash & Fold 5.0 kg -20%   Rp42,000   12 h
Subtotal                                 Rp136,500
TOTAL                                    Rp136,500
Everything ready in 72 h
```

Listing 7.33 does the same in Python by importing the classes from `laundry_pricing.py`, which must be in the same folder.

Listing 7.33 `shoe_promo_demo.py`: the same extension in Python.

```python
"""shoe_promo_demo.py: extending laundry_pricing.py without editing it."""
from laundry_pricing import ExpressService, Order, Service, WashService


class ShoeCleaningService(Service):
    def __init__(self, pairs):
        super().__init__("SH")
        if pairs <= 0:
            raise ValueError("pairs must be positive")
        self._pairs = pairs

    def name(self):
        return f"Shoe Cleaning {self._pairs} pair" + ("" if self._pairs == 1 else "s")

    def price(self):
        return self._pairs * 25000

    def ready_in_hours(self):
        return 72


# A promotion wraps any service, exactly as ExpressService does.
class PromoService(Service):
    def __init__(self, base, percent_off):
        super().__init__("P-" + base.code)
        if not 0 < percent_off < 100:
            raise ValueError("percent_off must be between 1 and 99")
        self._base = base
        self._percent_off = percent_off

    def name(self):
        return f"{self._base.name()} -{self._percent_off}%"

    def price(self):
        return self._base.price() - self._base.price() * self._percent_off // 100

    def ready_in_hours(self):
        return self._base.ready_in_hours()
```

Listing 7.33 (Continued) *shoe_promo_demo.py: the same extension in Python.*

```python
def main():
    order = Order("ORD-0713", "Budi Santoso", False)
    order.add(WashService(3.4))
    order.add(ShoeCleaningService(2))
    order.add(PromoService(ShoeCleaningService(1), 20))
    order.add(ExpressService(PromoService(WashService(5.0), 20)))
    order.print_receipt()


if __name__ == "__main__":
    main()
```

```
PROFITINA LAUNDRY  order ORD-0713  customer Budi Santoso
WF      Wash & Fold 3.4 kg                Rp24,500   48 h
SH      Shoe Cleaning 2 pairs             Rp50,000   72 h
P-SH    Shoe Cleaning 1 pair -20%         Rp20,000   72 h
EX-P-WF Express Wash & Fold 5.0 kg -20%   Rp42,000   12 h
Subtotal                                 Rp136,500
TOTAL                                    Rp136,500
Everything ready in 72 h
```

Check: 3.4 kg rounds up to 3.5 kg, 3.5 × 7000 = 24,500; two pairs are 50,000; one pair at 20% off is 25,000 − 5,000 = 20,000; 5.0 kg of Wash & Fold is 35,000, 20% off gives 28,000, and express

gives $\lfloor 3 \times 28{,}000/2 \rfloor = 42{,}000$, ready in $\max(6, 48/4) = 12$ hours. Subtotal 136,500; no member discount. Because $0.8 \times 1.5 = 1.5 \times 0.8$, promotion-inside-express and express-inside-promotion cost the same up to the rounding down of each step, at most one rupiah apart; for this order both give Rp42,000. Express of express is legal and applies Equation (7.9) twice: for 4.2 kg of Wash & Fold, 31,500 → 47,250 → 70,875, ready in $\max(6, 12/4) = 6$ hours. If the business does not want that, `ExpressService`'s constructor should reject a `base` that is already an `ExpressService`.

Solution 7.10. The output is:

```
['_Parent__secret', '_hint', '_Child__secret']
```

Inside `Parent.__init__`, `self.__secret` is mangled to `self._Parent__secret`; inside `Child.__init__`, the same text is mangled to `self._Child__secret`. They are two different attributes, so the child's assignment creates a new one and the parent's value 1 is untouched. This is exactly what `__name` is for: a subclass cannot clobber its parent's internals by choosing the same name. `_hint` is not mangled and is shared.

Solution 7.11. Callers of `WashService` may rely on properties such as "the price equals $7000 \cdot \hat{w}$" (Equation (7.8)) and "`chargeableKg()` is at least 3". Code that estimates machine loads by summing `chargeableKg()` over the day's orders would count every flat-rate bag as weighing nothing; code that audits prices against Equation (7.8) would flag every flat-rate bag. The property ϕ holds for `WashService` and fails for `FlatRateWash`, so by Equation (7.4) `FlatRateWash` is not a valid subtype of `WashService`. The better design makes `FlatRateWash` a sibling, a direct subclass of `Service` with its own `price()` returning 50,000 and its own `getName()`, promising nothing about weight.

Solution 7.12. A solution sketch in Python:

```
class Customer:
    def __init__(self, name, phone):
        self._phone = phone
        self.name = name            # goes through the setter

    @property
    def phone(self):
        return self._phone          # no setter: read-only

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        value = value.strip()
        if not value:
            raise ValueError("name must not be empty")
        self._name = value

    def __eq__(self, other):
        if not isinstance(other, Customer):
            return NotImplemented
        return self._phone == other._phone

    def __hash__(self):
        return hash(self._phone)
```

The phone must be read-only because it determines the hash. A set stores an object in the bucket chosen by its hash *at the time it was added*. If the phone could change afterwards, the object would sit in the wrong bucket: the set would no longer find it, and could even accept a second, "equal" customer. The name does not take part in equality or hashing, so it may safely change.

Solution 7.13. **Listing 7.34** and **Listing 7.35** are the two programs.

Listing 7.34 `ConstructorTrap.java`: a superclass constructor that calls an overridable method.

```
public class ConstructorTrap {
    public static void main(String[] args) {
        new Towel("white");
    }
}

class Item {
    Item() {
        System.out.println("Item(): " + describe());
    }

    String describe() {
        return "an item";
    }
}

class Towel extends Item {
    private final String color;

    Towel(String color) {
        super();
        this.color = color;
        System.out.println("Towel(): " + describe());
    }

    @Override
    String describe() {
        return "a " + color + " towel";
    }
}
```

```
Item(): a null towel
Towel(): a white towel
```

Listing 7.35 `constructor_trap.py`: the same trap in Python.

```python
"""constructor_trap.py: a constructor that calls an overridable method."""


class Item:
    def __init__(self):
        print("Item():", self.describe())

    def describe(self):
        return "an item"


class Towel(Item):
    def __init__(self, color):
        super().__init__()
        self.color = color
        print("Towel():", self.describe())

    def describe(self):
        return f"a {self.color} towel"


def main():
    try:
        Towel("white")
    except AttributeError as e:
        print("AttributeError:", e)


if __name__ == "__main__":
    main()
```

```
AttributeError: 'Towel' object has no attribute 'color'
```

Following **Figure 7.2**: `new Towel("white")` first runs `Item()`. Its call to `describe()` is dispatched on the object's dynamic type, which is already `Towel`, so `Towel.describe()` runs — but `Towel`'s constructor body has not yet assigned `color`, which still holds Java's default value `null`. Hence `a null towel`. Only after `Item()` returns does `Towel()` set the color, and its own call prints `a white towel`. Python behaves the same way, except that an attribute that has not been assigned does not exist at all, so the early call fails with `AttributeError`. In both languages the fix is the same: a constructor should not call methods that subclasses can override [6].

References

[1] O.-J. Dahl and K. Nygaard, "SIMULA: an ALGOL-based simulation language," *Communications of the ACM*, vol. 9, no. 9, pp. 671–678, 1966.

[2] Python Software Foundation, *The Python Language Reference*, version 3.14, and *The Python Standard Library*, module `abc`. [Online]. Available: https://docs.python.org/3/

[3] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*. Oracle America, 2023. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se21/html/

[4] OpenJDK, "JEP 513: Flexible constructor bodies," final in JDK 25, 2025. [Online]. Available: https://openjdk.org/jeps/513

[5] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, *The Java Virtual Machine Specification, Java SE 21 Edition*. Oracle America, 2023. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se21/html/

[6] J. Bloch, *Effective Java*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2018.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

[8] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in *Proc. OOPSLA '86*, Portland, OR, USA, 1986, pp. 38–45.

[9] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811–1841, 1994.

[10] L. Ramalho, *Fluent Python*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2022.

[11] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington, "A monotonic superclass linearization for Dylan," in *Proc. OOPSLA '96*, San Jose, CA, USA, 1996, pp. 69–82.

[12] M. Simionato, "The Python 2.3 method resolution order," Python Software Foundation, 2003. [Online]. Available: https://docs.python.org/3/howto/mro.html

[13] Oracle, "Controlling access to members of a class," *The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

[14] G. van Rossum, B. Warsaw, and A. Coghlan, "PEP 8 – Style guide for Python code," Python Enhancement Proposal 8, 2001. [Online]. Available: https://peps.python.org/pep-0008/

[15] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[16] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 1997.

Chapter 8

Evaluation and Discussion II: Recursive Structures, Robust Classes, and Class Hierarchies

This chapter adds no new language features. It works problems instead: twenty-one of them, every one solved from a blank page in both Java and Python. Fifteen come from one connected set of exercises on two recursive structures, a binary tree and an immutable list. Six more are new, and each targets one idea from Chapters 5 to 7: an order file read without crashing on bad lines, an array that grows itself and documents its own contract, an interface with two implementations that pass the same tests, a class hierarchy whose method calls you trace by hand, two duplicated game classes refactored into one hierarchy, and a class with four encapsulation bugs to find and fix. Every problem follows the same route: restate it in plain words, design it with a table or a UML drawing, write the Java and Python solutions, run the tests and show what they really print, explain the code line by line, study a tempting wrong solution and exactly how it fails, and finish with a *check yourself* variant and its answer. By the end of the chapter you will have a method for any problem that involves several cooperating classes, and you will have used it twenty-one times.

8.1 Where We Left Off

Chapters 5, 6 and 7 filled the toolbox that every later chapter uses. Let us list what is inside, because naming the right tool is half of solving a problem.

From **Chapter 5** we have *arrays* (a fixed number of cells of one type, numbered from 0), the growable `ArrayList` in Java and `list` in Python, reading input, *exceptions* (the `try`/`catch`/`finally` machinery in Java and `try`/`except`/`finally` in Python, checked and unchecked exceptions, and exceptions we define ourselves), and *files* (opening, reading line by line, and closing them safely with `try`-with-resources or `with`).

From **Chapter 6** we have *documentation* (Javadoc comments with `@param`, `@return` and `@throws`, and Python docstrings), *testing* (JUnit 6 in Java and `unittest` in Python: a test builds a situation, runs the code and states what the result must be), and *interfaces* (a named promise of methods with no fields of its own, implemented by any number of classes), together with *overloading* (several methods with one name and different parameter lists).

From **Chapter 7** we have *inheritance* (a subclass reuses and extends a superclass), *abstract classes*, *overriding* (a subclass replaces an inherited method), *polymorphism* (one call, `s.describe()`, runs different code depending on the object behind `s`), `super`, `final`, and the four *access levels* of Java (`private`, package-private, `protected`, `public`) with Python's naming conventions in their place.

This chapter uses all of them. It is organized in four layers.

1. **A method** (Section 8.2). Eight steps that take you from a paragraph of requirements to a set of classes, tests and code.
2. **Two small structures** (Section 8.3): a `Tree` class that does its work by *delegation*, asking its subtrees to do part of every job, and an immutable `List` built from two operations, `nil` and `cons`. Fifteen problems use them.
3. **Twenty-one worked problems.** Problems 8.1 to 8.7 (Section 8.4) are on trees, Problems 8.8 to 8.15 (Section 8.5) are on lists, and Problems 8.16 to 8.21 (Sections 8.6 to 8.8) are the new problems on exceptions, files, arrays, documentation, testing, interfaces, inheritance, polymorphism and access control.
4. **The usual closing sections**: a comparison of all the problems, where the ideas meet a real product, common mistakes, a summary, exercises and their full solutions.

Every problem is laid out in the same order, and each part has a bold label so that you can find it again: **The problem** (the statement), **In our own words**, **Design**, **The Java solution**, **The Python solution**, **Line by line**, **A tempting wrong solution**, and **Check yourself**. The tests are built into the programs, so the output that follows each listing *is* the test report.

> **HOW TO USE THIS CHAPTER.** Read the statement of a problem, then cover the rest with a sheet of paper and spend ten minutes on it yourself: draw the design, write the base case, try one example by hand. A solution you have struggled with first is understood in a way that a solution you have only read never is. Then read on, and compare. The *check yourself* variants tell you whether the idea has really arrived.

A word about the tests. JUnit 6 is the standard testing tool for Java (Chapter 6), but it is a separate library that has to be downloaded. To keep every Java program in this chapter runnable with nothing more than a JDK, each one tests itself with a small helper method, `check`, that compares an expected value with the actual one, prints `PASS` and a name when they agree, and throws an `AssertionError` when they do not. Section 8.7 shows how the same checks look as JUnit 6 tests and how to run them. The Python programs use `unittest` directly, because it ships with Python [1].

8.2 How to Approach a Multi-Class Design Problem

Most problems in object-oriented programming are not “write this one method”. They are “here is a situation with several kinds of things in it; make the program”. A laundry takes orders from customers, each order uses a service, each service has a price rule, a courier delivers the finished order. Where do you start? This section gives a method, eight steps that tell you what to write on paper, in what order, before you write a line of code. On a small problem they take ten minutes; on a large one they separate a design from a pile of classes.

> **DEFINITION.** A *responsibility* is something a class knows (a field) or does (a method). A good class has a small set of responsibilities that belong together, and it is the only class that changes its own fields.

Step 1 — Underline the nouns and circle the verbs. Read the requirements once, slowly, with a pen. Nouns (*order*, *customer*, *service*, *weight*, *price*) are candidate classes and fields; verbs (*weigh*, *price*, *deliver*, *parse*) are candidate methods. Do not decide anything yet; just collect.

Step 2 — Sort the nouns. A noun that has its own data *and* its own behavior becomes a class (`Order`). A noun that is only a value becomes a field (`kg` is a `double` inside `Order`). A noun that names a *kind* of something (*express service*, *standard service*) is a hint of an interface or a class hierarchy.

Step 3 — Give every verb an owner. For each method, ask "which object has the data this needs?" and put the method there. Pricing needs the rate, so it belongs to the price rule, not to the customer. This single question, asked honestly, prevents most bad designs.

Step 4 — Choose the relationships. Four kinds cover almost everything: *has-a* (an order has a list of items: a field), *uses* (a parser uses an order: a parameter or local variable), *is-a* with shared code (an express service is a wash service: inheritance, Chapter 7), and *can-do* with no shared code (anything that can price a weight: an interface, Chapter 6). Prefer has-a and can-do; use is-a only when the subclass really is a special case and really shares code [2].

Step 5 — Decide the access level of everything. Fields are `private` (in Python, named with a leading underscore). A method is `public` only if a class outside needs it; `protected` if only subclasses need it; otherwise package-private or `private`. Ask of every field: "who may change this, and what rule must hold after the change?" The answer tells you which checks the constructor and the setters must make.

Step 6 — Draw it. A UML class diagram [3] is a box per class with three compartments (name, fields, methods), a sign before each member (`-` private, `#` protected, `~` package-private, `+` public), a hollow triangle for "extends", a dashed line with a hollow triangle for "implements", and a dashed arrow for "uses". For a small problem a table with the same information works just as well; this chapter uses both.

Step 7 — Write the tests before the code. For each method write two or three calls with their expected results, and always include the *edge cases*: the empty list, the leaf, the missing file, the zero weight, the index equal to the size. Most wrong solutions in this chapter pass the ordinary example and fail exactly one edge case.

Step 8 — Implement the smallest piece, run, repeat. Write one method, run its tests, then the next. When the tests pass, read the code once more looking only at duplication and at access levels, and tidy it (refactoring, Problem 8.20).

Table 8.1 condenses the method into one page. You will see it applied, explicitly, in every problem of this chapter.

Table 8.1 The eight-step method for a multi-class design problem. Steps 1 to 6 are design, Step 7 is specification, and Step 8 is implementation.

Step	Write on paper	Question it answers
1	Nouns and verbs of the requirements	What is in the situation?
2	Classes, fields, and "kinds of"	Which nouns deserve a class?
3	An owner for every method	Who has the data this needs?
4	has-a, uses, is-a, can-do	How are the classes connected?
5	An access level for every member	Who may change what, under which rule?
6	A UML diagram or a design table	Does the whole design fit on one page?
7	Tests, with the edge cases	How will I know it works?
8	One method at a time, then tidy	Does it work now? Is it clean?

For recursive methods, which fill Sections 8.4 and 8.5, Steps 3 and 7 take a special shape that is worth stating on its own.

> **THE DELEGATION RECIPE FOR RECURSION.** To write a recursive method on a tree or a list, answer three questions in writing. (1) *Base case*: what is the answer for the smallest structure, the empty list or the missing subtree? (2) *Delegation*: what do I ask my smaller parts (the tail,

> the left and right subtrees) to compute for me? Assume they do it correctly. (3) *My own job*: how do I combine their answers with my own data (the head, the root)? If the three answers are right, the method is right, by induction on the size of the structure.

The recipe is exactly the reasoning of mathematical induction [4]: the base case is the base of the induction, and "assume the smaller parts do it correctly" is the induction hypothesis. The design tables of Sections 8.4 and 8.5 have one column for each of the three answers.

8.3 The Two Structures of This Chapter: Trees and Cons Lists

Fifteen of the problems below use two small data structures. Both are *recursive*: a tree contains smaller trees, and a list contains a smaller list. Both are also excellent practice for Chapters 5 to 7, because they force you to think about references (who points at what), about access control (who may change what), about exceptions (what happens when you ask the empty list for its first element), and about testing (how do you know a recursive method is right?).

8.3.1 A binary tree that does its work by delegation

A *binary tree* is either missing (in code, `null` in Java and `None` in Python) or a *node* that holds a value, called its *root*, and two binary trees, called its *left subtree* and its *right subtree*. A node whose two subtrees are both missing is a *leaf*. **Figure 8.1** shows a binary tree with nine nodes: the root is 1, the leaves are 5, 6, 8 and 9, and node 4 has a right subtree but no left one.

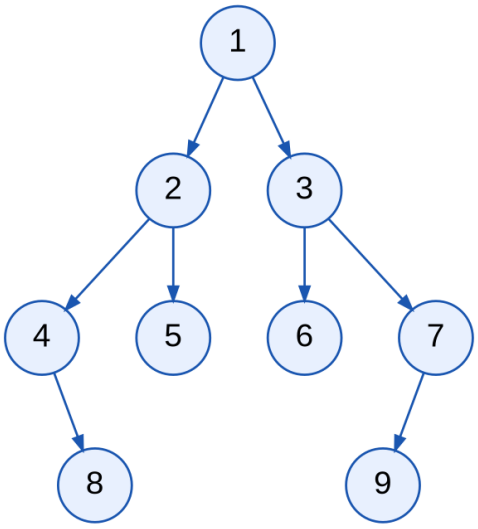


Figure 8.1 The binary tree of Problem 8.1. Node 4 has only a right child (8) and node 7 has only a left child (9); every other inner node has two children.

Listing 8.1 is the `Tree` class that all seven tree problems start from. It is a *nested* class: it sits inside the public class `TreeStart`, marked `static` so that a `Tree` does not need a `TreeStart` object to exist. Each program of Section 8.4 carries its own copy of this nested class, with only the methods that its problem needs, so that every file compiles and runs on its own.

Listing 8.1 `TreeStart.java`: the starting `Tree` class with two constructors, `mirror()` by delegation, a compact `toString()` and a sideways drawing.

```java
// TreeStart.java - the starting Tree class: two constructors,
// mirror() by delegation, and two ways to print a tree.
public class TreeStart {

    /** A binary tree whose nodes are labelled by integers. */
    static final class Tree {
        private int root;
        private Tree left, right;

        /** A branch: a root with (possibly empty) subtrees. */
        Tree(int root, Tree left, Tree right) {
            this.root = root;
            this.left = left;
            this.right = right;
        }

        /** A leaf: a special case of a branch. */
        Tree(int root) {
            this(root, null, null);
        }

        /** Mirror myself: my subtrees mirror themselves, then I swap them. */
        void mirror() {
            if (left != null) {
                left.mirror();          // delegation to the left subtree
            }
            if (right != null) {
                right.mirror();         // delegation to the right subtree
            }
            Tree originalLeft = left;   // my own job: swap the branches
            left = right;
            right = originalLeft;
        }

        /** Compact form: a leaf is its number, a branch is root(left,right). */
        @Override
        public String toString() {
            if (left == null && right == null) {
                return Integer.toString(root);
            }
            String l = (left == null) ? "-" : left.toString();
            String r = (right == null) ? "-" : right.toString();
            return root + "(" + l + "," + r + ")";
        }
```

Listing 8.1 (Continued) *TreeStart.java: the starting Tree class with two constructors, mirror() by delegation, a compact toString() and a sideways drawing.*

```
        /** Sideways drawing: right subtree on top, 4 spaces per level. */
        String toStringAdv() {
            return toStringFrom(0);
        }

        private String toStringFrom(int depth) {
            String sRight = (right == null) ? "" : right.toStringFrom(depth + 4);
            String sLeft = (left == null) ? "" : left.toStringFrom(depth + 4);
            return sRight + " ".repeat(depth) + root + "\n" + sLeft;
        }
    }

    public static void main(String[] args) {
        Tree t = new Tree(3,
                new Tree(4, new Tree(6), new Tree(7)),
                new Tree(5, new Tree(8), new Tree(9)));
        System.out.println("t          = " + t);
        System.out.print(t.toStringAdv());
        t.mirror();
        System.out.println("mirrored t = " + t);
        System.out.print(t.toStringAdv());
    }
}
```

```
t          = 3(4(6,7),5(8,9))
        9
    5
        8
3
        7
    4
        6
mirrored t = 3(5(9,8),4(7,6))
        6
    4
        7
3
        8
    5
        9
```

Three details deserve a slow reading. First, the fields `root`, `left` and `right` are `private`: only code inside `Tree` can read or change them, which is why every operation on a tree is a method of `Tree`. Second, the leaf constructor `Tree(int root)` does not repeat the three assignments; it calls the branch constructor with `this(root, null, null)`, the constructor chaining of Chapter 7. Third, look at `mirror()`. It does not walk the tree with a loop. It asks its left subtree to mirror itself, asks its right subtree to mirror itself, and then does its own small job, swapping its two references. That is the *delegation recipe* of Section 8.2: the base case is hidden in the `if (left != null)` tests (a missing subtree has nothing to mirror), the delegation is the two recursive calls, and the node's own job is the swap.

The two printing methods are helpers for seeing results. `toString()` writes a leaf as its number and a branch as `root(left,right)`, with `-` for a missing subtree, so the tree of **Figure 8.1** prints as `1(2(4(-,8),5),3(6,7(9,-)))`. `toStringAdv()` draws the tree on its side: turn your head to the left and the right subtree is on top. Every test in Section 8.4 compares the `toString()` form with an expected string, because two trees with the same string have exactly the same shape and values.

Listing 8.2 is the same class in Python.

Listing 8.2 `tree_start.py`: the same `Tree` class in Python. The leading underscore marks an attribute as private by convention.

```
# tree_start.py - the starting Tree class: mirror() by delegation,
# and two ways to print a tree.


class Tree:
    """A binary tree whose nodes are labelled by integers."""

    def __init__(self, root, left=None, right=None):
        self._root = root          # leading underscore: "private by convention"
        self._left = left
        self._right = right

    def mirror(self):
        """Mirror myself: my subtrees mirror themselves, then I swap them."""
        if self._left is not None:
            self._left.mirror()    # delegation to the left subtree
        if self._right is not None:
            self._right.mirror()   # delegation to the right subtree
        self._left, self._right = self._right, self._left

    def __str__(self):
        """Compact form: a leaf is its number, a branch is root(left,right)."""
        if self._left is None and self._right is None:
            return str(self._root)
        l = "-" if self._left is None else str(self._left)
        r = "-" if self._right is None else str(self._right)
        return f"{self._root}({l},{r})"

    def to_string_adv(self, depth=0):
        """Sideways drawing: right subtree on top, 4 spaces per level."""
        s_right = "" if self._right is None else self._right.to_string_adv(depth + 4)
        s_left = "" if self._left is None else self._left.to_string_adv(depth + 4)
        return s_right + " " * depth + str(self._root) + "\n" + s_left


if __name__ == "__main__":
    t = Tree(3, Tree(4, Tree(6), Tree(7)), Tree(5, Tree(8), Tree(9)))
    print("t          =", t)
    print(t.to_string_adv(), end="")
    t.mirror()
    print("mirrored t =", t)
    print(t.to_string_adv(), end="")
```

```
t          = 3(4(6,7),5(8,9))
        9
    5
        8
3
        7
    4
        6
mirrored t = 3(5(9,8),4(7,6))
        6
    4
        7
3
        8
    5
        9
```

Python has no `private` keyword. The leading underscore in `self._root` is a *convention* that says "this is an implementation detail; do not touch it from outside the class". Nothing stops a determined programmer, but every Python programmer reads the underscore as a fence. Python also has no overloaded constructors; the default arguments `left=None, right=None` let one `__init__` serve both as the branch constructor and as the leaf constructor. The swap in `mirror()` is a single *tuple assignment*, `self._left, self._right = self._right, self._left`, which evaluates the whole right-hand side before assigning anything. The two programs print exactly the same lines.

8.3.2 An immutable list built from nil and cons

The second structure is a list of integers defined the way a mathematician would define it. A list is either *empty*, called `nil`, or it is built by `cons` from a first element, called the *head*, and a shorter list, called the *tail*. The list $[2, -5, 8]$ is therefore

$$[2, -5, 8] = \text{cons}(2, \text{cons}(-5, \text{cons}(8, \text{nil}))) \tag{8.1}$$

and **Figure 8.2** draws the objects that Equation (8.1) creates: three non-empty cells, each holding a head and a reference to its tail, and one shared empty list at the end. The names `cons` and `nil` come from the Lisp family of languages, where this definition was born [4].

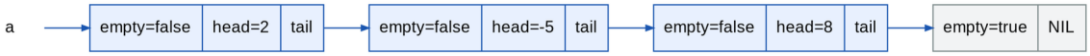


Figure 8.2 The objects behind `List a = cons(2, cons(-5, cons(8, nil())))`. Each cell has a head and a tail reference; the last tail is the single shared empty list.

The Java version is split into three files, exactly as a real project would split it: `List` itself (**Listing 8.3**), the exception it throws (**Listing 8.4**), and a utility class of helper operations (**Listing 8.5**). **Figure 8.3** shows how they fit together.

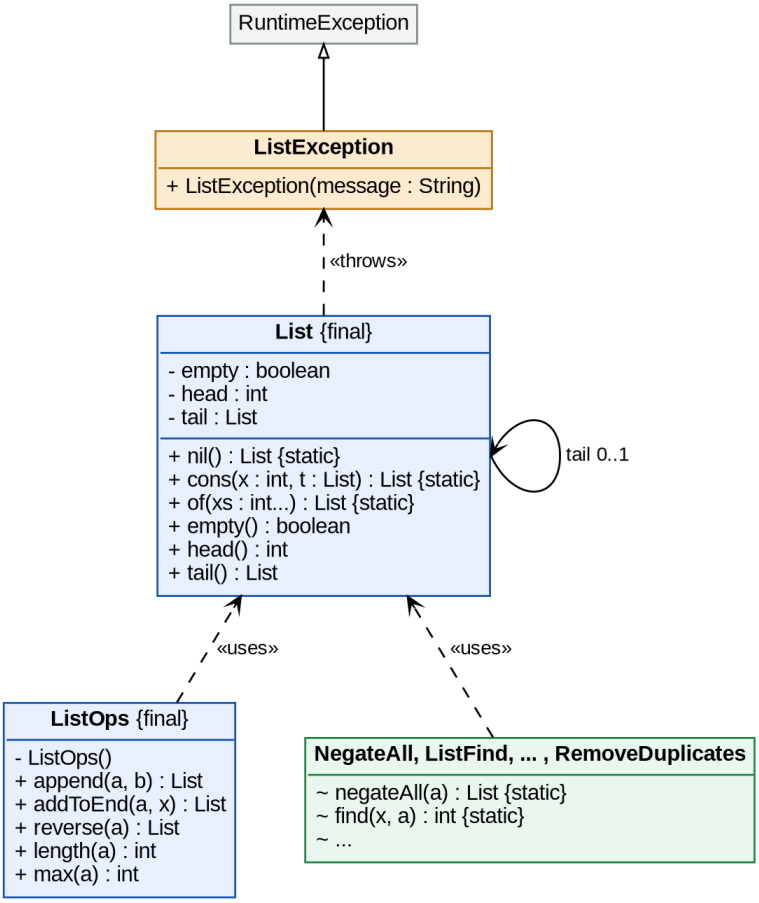


Figure 8.3 UML class diagram of the list classes. `ListException` is unchecked because it extends `RuntimeException`; every problem class of Section 8.5 only uses `List`, it does not extend it.

Listing 8.3 `List.java`: an immutable list of integers. The constructors are private, so `nil()`, `cons()` and `of()` are the only ways to build a list.

```java
// List.java - an immutable list of integers, built from "nil" and "cons".
public final class List {

    private final boolean empty;
    private final int head;
    private final List tail;

    private static final List NIL = new List();     // one shared empty list

    private List(int head, List tail) {              // a non-empty list
        this.empty = false;
        this.head = head;
        this.tail = tail;
    }

    private List() {                                 // the empty list
        this.empty = true;
        this.head = 0;
        this.tail = null;
    }

    /** The empty list, written [] in the comments of this chapter. */
    public static List nil() {
        return NIL;
    }

    /** A new list whose first element is x and whose rest is t. */
    public static List cons(int x, List t) {
        return new List(x, t);
    }

    /** Convenience: List.of(2, -5, 8) is cons(2, cons(-5, cons(8, nil()))). */
    public static List of(int... xs) {
        List result = nil();
        for (int i = xs.length - 1; i >= 0; i--) {  // build from the back
            result = cons(xs[i], result);
        }
        return result;
    }

    public boolean empty() {
        return empty;
    }
```

Listing 8.3 (Continued) *List.java: an immutable list of integers. The constructors are private, so nil(), cons() and of() are the only ways to build a list.*

```
    public int head() {
        if (empty) {
            throw new ListException("head() of the empty list");
        }
        return head;
    }

    public List tail() {
        if (empty) {
            throw new ListException("tail() of the empty list");
        }
        return tail;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("[");
        for (List p = this; !p.empty; p = p.tail) {
            sb.append(p.head);
            if (!p.tail.empty) {
                sb.append(", ");
            }
        }
        return sb.append("]").toString();
    }
}
```

Listing 8.4 `ListException.java`: an unchecked exception for operations that need a non-empty list.

```
// ListException.java - thrown when an operation needs a non-empty list.
public class ListException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public ListException(String message) {
        super(message);
    }
}
```

Listing 8.5 `ListOps.java`: helper operations written recursively. The private constructor stops anyone from creating a `ListOps` object.

```
// ListOps.java - recursive helper operations on List (a utility class).
public final class ListOps {

    private ListOps() {                // no objects: only static methods
    }

    /** All of a, followed by all of b. */
    public static List append(List a, List b) {
        if (a.empty()) {
            return b;
        }
        return List.cons(a.head(), append(a.tail(), b));
    }

    /** A copy of a with x added at the end. */
    public static List addToEnd(List a, int x) {
        return append(a, List.cons(x, List.nil()));
    }

    /** The elements of a in reverse order. */
    public static List reverse(List a) {
        if (a.empty()) {
            return List.nil();
        }
        return addToEnd(reverse(a.tail()), a.head());
    }

    /** The number of elements of a. */
    public static int length(List a) {
        return a.empty() ? 0 : 1 + length(a.tail());
    }

    /** The largest element; an empty list has none, so that is an error. */
    public static int max(List a) {
        if (a.empty()) {
            throw new ListException("max() of the empty list");
        }
        if (a.tail().empty()) {
            return a.head();
        }
        return Math.max(a.head(), max(a.tail()));
    }
}
```

The design choices in these three files are all Chapter 5 to 7 decisions, and they are worth naming one by one.

- **Immutability.** The three fields of `List` are `private final`. Once a list exists it never changes; every operation that "changes" a list builds a new one. That is why `negateAll` in Problem 8.8 must *return* a list, and why sharing tails between lists (as `append` does with `b`) is perfectly safe.
- **Private constructors, public factories.** Nobody outside `List` can call `new List(...)`. They must use `nil()`, `cons()` or `of()`. The empty list is created once, in the `static final` field `NIL`, and shared; there is no reason to have two empty lists.
- **An exception for a broken contract.** `head()` and `tail()` of the empty list have no sensible answer, so they throw `ListException`. Because it extends `RuntimeException`, it is *unchecked*: the compiler does not force every caller to write `try`, which suits an error that correct code never triggers.
- **A utility class, not a superclass.** `ListOps` holds only `static` methods and has a `private` constructor. The problem classes of Section 8.5 call `List.cons(...)` and `SkipElement.skip(...)` by name. It would be tempting to write `class NegateAll extends List` just to be able to type `cons(...)` without the `List.` prefix, but that is inheritance for

convenience: it claims that a `NegateAll` *is a* list, which is false. Chapter 7 reserved inheritance for real is-a relationships, and `List` is `final` so the question does not even arise.

Listing 8.6 takes the classes for a first drive.

Listing 8.6 `ListStart.java`: a first tour of `List`, `ListOps` and `ListException`.

```
// ListStart.java - a first tour of List, ListOps and ListException.
public class ListStart {

    public static void main(String[] args) {
        List a = List.cons(2, List.cons(-5, List.cons(8, List.nil())));
        List b = List.of(7, 1);
        System.out.println("a                = " + a);
        System.out.println("a.head()         = " + a.head());
        System.out.println("a.tail()         = " + a.tail());
        System.out.println("append(a, b)     = " + ListOps.append(a, b));
        System.out.println("addToEnd(a, 0)   = " + ListOps.addToEnd(a, 0));
        System.out.println("reverse(a)       = " + ListOps.reverse(a));
        System.out.println("length(a)        = " + ListOps.length(a));
        System.out.println("max(a)           = " + ListOps.max(a));
        System.out.println("a is unchanged   = " + a);
        try {
            ListOps.max(List.nil());
        } catch (ListException e) {
            System.out.println("caught ListException: " + e.getMessage());
        }
    }
}
```

Compile the four files together and run the demo: `javac List.java ListException.java ListOps.java ListStart.java`, then `java ListStart`.

```
a                = [2, -5, 8]
a.head()         = 2
a.tail()         = [-5, 8]
append(a, b)     = [2, -5, 8, 7, 1]
addToEnd(a, 0)   = [2, -5, 8, 0]
reverse(a)       = [8, -5, 2]
length(a)        = 3
max(a)           = 8
a is unchanged   = [2, -5, 8]
caught ListException: max() of the empty list
```

Notice the line `a is unchanged = [2, -5, 8]`: `append`, `addToEnd` and `reverse` all returned new lists and left `a` alone. Notice also the last line. `max` of the empty list threw a `ListException`, and the `catch` block in `main` caught it and printed its message; without the `try`, the program would have stopped with a stack trace.

The Python version keeps the class and the helper functions in one *module*, `cons_list.py` (**Listing 8.7**), because Python programmers group related definitions by file rather than by class. **Listing 8.8** is the same tour.

Listing 8.7 `cons_list.py`: the list class, its exception and the helper operations in one Python module.

```
# cons_list.py - an immutable list of integers built from nil() and cons(),
# with the helper operations of ListOps as plain functions.


class ListException(Exception):
    """Raised when an operation needs a non-empty list."""


class List:
    __slots__ = ("_empty", "_head", "_tail")        # no other attributes allowed

    def __init__(self, empty, head=0, tail=None):
        self._empty = empty
        self._head = head
        self._tail = tail

    @staticmethod
    def nil():
        """The empty list, written [] in the comments of this chapter."""
        return _NIL

    @staticmethod
    def cons(x, t):
        """A new list whose first element is x and whose rest is t."""
        return List(False, x, t)

    @staticmethod
    def of(*xs):
        """Convenience: List.of(2, -5, 8) is cons(2, cons(-5, cons(8, nil()))))."""
        result = List.nil()
        for x in reversed(xs):                      # build from the back
            result = List.cons(x, result)
        return result

    def empty(self):
        return self._empty

    def head(self):
        if self._empty:
            raise ListException("head() of the empty list")
        return self._head
```

Listing 8.7 (Continued) *cons_list.py: the list class, its exception and the helper operations in one Python module.*

```
    def tail(self):
        if self._empty:
            raise ListException("tail() of the empty list")
        return self._tail

    def __str__(self):
        items = []
        p = self
        while not p._empty:
            items.append(str(p._head))
            p = p._tail
        return "[" + ", ".join(items) + "]"


_NIL = List(True)                                   # one shared empty list


def append(a, b):
    """All of a, followed by all of b."""
    if a.empty():
        return b
    return List.cons(a.head(), append(a.tail(), b))


def add_to_end(a, x):
    """A copy of a with x added at the end."""
    return append(a, List.cons(x, List.nil()))


def reverse(a):
    """The elements of a in reverse order."""
    if a.empty():
        return List.nil()
    return add_to_end(reverse(a.tail()), a.head())


def length(a):
    """The number of elements of a."""
    return 0 if a.empty() else 1 + length(a.tail())
```

Listing 8.7 (Continued) *cons_list.py: the list class, its exception and the helper operations in one Python module.*

```
def max_of(a):
    """The largest element; an empty list has none, so that is an error."""
    if a.empty():
        raise ListException("max() of the empty list")
    if a.tail().empty():
        return a.head()
    return max(a.head(), max_of(a.tail()))
```

Listing 8.8 `list_start.py`: a first tour of the Python list module.

```
# list_start.py - a first tour of List, the helper functions and ListException.
from cons_list import List, ListException, append, add_to_end, reverse, length, max_of

a = List.cons(2, List.cons(-5, List.cons(8, List.nil())))
b = List.of(7, 1)
print("a                =", a)
print("a.head()         =", a.head())
print("a.tail()         =", a.tail())
print("append(a, b)     =", append(a, b))
print("add_to_end(a, 0) =", add_to_end(a, 0))
print("reverse(a)       =", reverse(a))
print("length(a)        =", length(a))
print("max_of(a)        =", max_of(a))
print("a is unchanged   =", a)
try:
    max_of(List.nil())
except ListException as e:
    print("caught ListException:", e)
```

```
a                = [2, -5, 8]
a.head()         = 2
a.tail()         = [-5, 8]
append(a, b)     = [2, -5, 8, 7, 1]
add_to_end(a, 0) = [2, -5, 8, 0]
reverse(a)       = [8, -5, 2]
length(a)        = 3
max_of(a)        = 8
a is unchanged   = [2, -5, 8]
caught ListException: max() of the empty list
```

Two Python details differ from Java. `__slots__` declares the only three attribute names an object may have, so a typo such as `a._haed = 3` raises an error instead of silently creating a new attribute. And the helper for the maximum is called `max_of`, because `max` is a built-in Python function that we do not want to hide. In Python every exception is unchecked, so `ListException` simply extends `Exception`.

With both structures in hand, we can start solving.

8.4 Problems on Binary Trees

The seven problems of this section all extend the `Tree` class of Section 8.3.1. Problem 8.1 builds a tree by hand; Problems 8.2 to 8.5 each add a recursive method that visits every node; Problems 8.6 and 8.7 add methods for a *binary search tree*, a tree whose values are arranged so that a search needs to visit only one path.

8.4.1 Problem 8.1 — Building a tree by hand

> **PROBLEM.** Write Java code to create the tree of **Figure 8.1** using `new Tree` statements. You will need nine such statements.

In our own words. We have two constructors, `Tree(int root)` for a leaf and `Tree(int root, Tree left, Tree right)` for a branch. Using each `new` exactly once per node, we must produce an object structure whose shape is the drawing: 1 at the top, 2 and 3 below it, 4 and 5 below 2, 6 and 7 below 3, 8 hanging to the *right* of 4, and 9 hanging to the *left* of 7.

Design. The key observation is about *order*. The branch constructor needs its two subtrees as arguments, so a subtree must exist before its parent can be built. The tree must therefore be built *bottom-up*: leaves first, then their parents, then the root. **Table 8.2** lists the nine statements in a valid order, with the level of each node counted from the root at level 0.

Table 8.2 The nine constructor calls of Problem 8.1 in bottom-up order. A node can be built only after both of its subtrees exist.

#	Node	Level	Left subtree	Right subtree	Constructor
1–4	5, 6, 8, 9	2, 2, 3, 3	none	none	`Tree(int)`
5	4	2	none	8	`Tree(4, null, leaf8)`
6	7	2	9	none	`Tree(7, leaf9, null)`
7	2	1	4	5	`Tree(2, t4, leaf5)`
8	3	1	6	7	`Tree(3, leaf6, t7)`
9	1	0	2	3	`Tree(1, t2, t3)`

The Java solution. **Listing 8.9** builds the tree in `build()`, checks its shape and its size, and also builds the tempting wrong version discussed below.

Listing 8.9 `BuildTree.java`: the nine `new Tree` statements, a recursive `size()`, and self-checks.

```java
// BuildTree.java - Problem 8.1: build a nine-node tree with nine "new Tree" statements.
public class BuildTree {

    static final class Tree {
        private int root;
        private Tree left, right;

        Tree(int root, Tree left, Tree right) {
            this.root = root;
            this.left = left;
            this.right = right;
        }

        Tree(int root) {
            this(root, null, null);
        }

        /** Number of nodes: 1 for me, plus whatever my subtrees report. */
        int size() {
            int l = (left == null) ? 0 : left.size();
            int r = (right == null) ? 0 : right.size();
            return 1 + l + r;
        }

        @Override
        public String toString() {
            if (left == null && right == null) {
                return Integer.toString(root);
            }
            String l = (left == null) ? "-" : left.toString();
            String r = (right == null) ? "-" : right.toString();
            return root + "(" + l + "," + r + ")";
        }
    }

    /** Builds the tree bottom-up: children must exist before their parent. */
    static Tree build() {
        Tree leaf5 = new Tree(5);              // 1
        Tree leaf6 = new Tree(6);              // 2
        Tree leaf8 = new Tree(8);              // 3
        Tree leaf9 = new Tree(9);              // 4
        Tree t4 = new Tree(4, null, leaf8);    // 5: 8 hangs to the RIGHT of 4
        Tree t7 = new Tree(7, leaf9, null);    // 6: 9 hangs to the LEFT of 7
        Tree t2 = new Tree(2, t4, leaf5);      // 7
        Tree t3 = new Tree(3, leaf6, t7);      // 8
        return new Tree(1, t2, t3);            // 9
```

Listing 8.9 (Continued) *BuildTree.java: the nine new Tree statements, a recursive size(), and self-checks.*

```
    }

    /** Tempting but wrong: every lone child placed on the left. */
    static Tree buildWrong() {
        Tree t4 = new Tree(4, new Tree(8), null);
        Tree t7 = new Tree(7, new Tree(9), null);
        return new Tree(1, new Tree(2, t4, new Tree(5)), new Tree(3, new Tree(6), t7));
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        Tree t = build();
        System.out.println("t       = " + t);
        System.out.println("wrong t = " + buildWrong());
        check("shape", "1(2(4(-,8),5),3(6,7(9,-)))", t.toString());
        check("size", 9, t.size());
        check("wrong shape differs", false, t.toString().equals(buildWrong().toString()));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
t       = 1(2(4(-,8),5),3(6,7(9,-)))
wrong t = 1(2(4(8,-),5),3(6,7(9,-)))
PASS shape
PASS size
PASS wrong shape differs
All 3 checks passed.
```

The Python solution. **Listing 8.10** is the same program. The tests are methods of a `unittest.TestCase` class whose names begin with `test_`; `unittest.main(verbosity=2)` finds and runs them and prints one line per test.

Listing 8.10 `build_tree.py`: the same tree in Python, tested with `unittest`.

```
# build_tree.py - Problem 8.1: build a nine-node tree with nine Tree(...) calls.
import sys
import unittest


class Tree:
    def __init__(self, root, left=None, right=None):
        self._root = root
        self._left = left
        self._right = right

    def size(self):
        """Number of nodes: 1 for me, plus whatever my subtrees report."""
        l = 0 if self._left is None else self._left.size()
        r = 0 if self._right is None else self._right.size()
        return 1 + l + r

    def __str__(self):
        if self._left is None and self._right is None:
            return str(self._root)
        l = "-" if self._left is None else str(self._left)
        r = "-" if self._right is None else str(self._right)
        return f"{self._root}({l},{r})"


def build():
    """Builds the tree bottom-up: children must exist before their parent."""
    leaf5 = Tree(5)                    # 1
    leaf6 = Tree(6)                    # 2
    leaf8 = Tree(8)                    # 3
    leaf9 = Tree(9)                    # 4
    t4 = Tree(4, None, leaf8)          # 5: 8 hangs to the RIGHT of 4
    t7 = Tree(7, leaf9, None)          # 6: 9 hangs to the LEFT of 7
    t2 = Tree(2, t4, leaf5)            # 7
    t3 = Tree(3, leaf6, t7)            # 8
    return Tree(1, t2, t3)             # 9


def build_wrong():
    """Tempting but wrong: every lone child placed on the left."""
    t4 = Tree(4, Tree(8))
    t7 = Tree(7, Tree(9))
    return Tree(1, Tree(2, t4, Tree(5)), Tree(3, Tree(6), t7))
```

Listing 8.10 (Continued) *build_tree.py: the same tree in Python, tested with unittest.*

```
class TestBuild(unittest.TestCase):
    def test_shape(self):
        self.assertEqual("1(2(4(-,8),5),3(6,7(9,-)))", str(build()))

    def test_size(self):
        self.assertEqual(9, build().size())

    def test_wrong_shape_differs(self):
        self.assertNotEqual(str(build()), str(build_wrong()))


if __name__ == "__main__":
    print("t       =", build())
    print("wrong t =", build_wrong())
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
t       = 1(2(4(-,8),5),3(6,7(9,-)))
wrong t = 1(2(4(8,-),5),3(6,7(9,-)))
test_shape (__main__.TestBuild.test_shape) ... ok
test_size (__main__.TestBuild.test_size) ... ok
test_wrong_shape_differs (__main__.TestBuild.test_wrong_shape_differs) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Line by line. Statements 1 to 4 create the four leaves; nothing points at them yet. Statement 5, `new Tree(4, null, leaf8)`, is the one to read twice: the *second* argument is the left subtree, and it is `null`, because in the drawing 8 hangs off the right-hand edge of 4. Statement 6 is its mirror image: 9 is the *left* child of 7, so `leaf9` is the second argument and `null` the third. Statements 7 and 8 build the two middle nodes from objects that now exist, and statement 9 builds the root and returns it. `size()` is our first delegation method: a node reports 1 for itself plus whatever its subtrees report, and a missing subtree contributes 0. The check `size` equal to 9 confirms that no node was forgotten and none was built twice. (The Python test lines read `test_shape (__main__.TestBuild.test_shape)` on Python 3.11 and later; Python 3.10 prints the shorter form `test_shape (__main__.TestBuild)`.)

A tempting wrong solution. In a hand drawing, a single child often looks as if it hangs "below" its parent, and the easiest constructor call is `new Tree(4, new Tree(8), null)`: the child goes in the first slot. The program shows the result: `wrong t = 1(2(4(8,-),5),3(6,7(9,-)))`. Node 8 is now a *left* child. The tree still has nine nodes, so a test on the size alone would pass. Only a test on the exact shape catches it, which is why the check `shape` compares the full string. The lesson generalizes: for a tree, the left–right order is part of the data.

> **CHECK YOURSELF.** Write the `toString()` form of the tree with root 10, whose left subtree is a leaf 20 and whose right subtree is 30 with only a left child 40. How many `new` statements are needed, and in which order?
>
> *Answer.* `10(20,30(40,-))`. Four statements: the leaves 20 and 40 first (in either order), then `new Tree(30, leaf40, null)`, then `new Tree(10, leaf20, t30)`.

8.4.2 Problem 8.2 — Tripling every node

> **PROBLEM.** Include a recursive method `public void triple()` in `Tree` that multiplies every node of the tree by three. For example, the tree with root 3, children 4 and 5, and grandchildren 6, 7, 8, 9 becomes the tree with root 9, children 12 and 15, and grandchildren 18, 21, 24, 27. Hint: ask your left and right subtrees to apply the procedure to themselves, then multiply your root by three.

In our own words. `triple()` changes the tree in place; it returns nothing (`void`). After the call, every value is three times what it was, and the shape is unchanged. **Figure 8.4** shows the example.

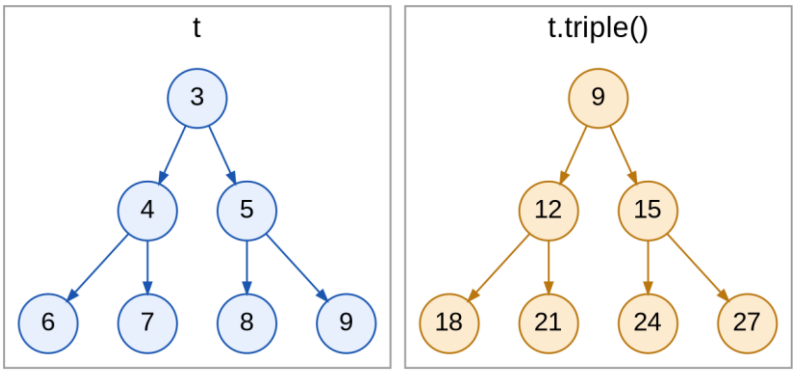


Figure 8.4 The sample tree before and after `t.triple()`: the shape is the same and every value is multiplied by three.

Design. The three questions of the delegation recipe have short answers (**Table 8.3**).

Table 8.3 The delegation recipe applied to `triple()`.

Question	Answer for `triple()`
Base case	A missing subtree: do nothing (guarded by `if (left != null)`)
Delegation	`left.triple()` and `right.triple()`
My own job	`root = 3 * root`
Returns	nothing; the tree is changed in place

The Java solution. **Listing 8.11** gives the method, a wrong variant, and the checks.

Listing 8.11 `TripleTree.java`: tripling every node by delegation, with a crashing wrong variant.

```java
// TripleTree.java - Problem 8.2: multiply every node of a tree by three.
public class TripleTree {

    static final class Tree {
        private int root;
        private Tree left, right;

        Tree(int root, Tree left, Tree right) {
            this.root = root;
            this.left = left;
            this.right = right;
        }

        Tree(int root) {
            this(root, null, null);
        }

        /** Triple every node: subtrees first (by delegation), then my root. */
        void triple() {
            if (left != null) {
                left.triple();
            }
            if (right != null) {
                right.triple();
            }
            root = 3 * root;
        }

        /** Tempting but wrong: forgets that a subtree may be missing. */
        void tripleWrong() {
            root = 3 * root;
            left.tripleWrong();
            right.tripleWrong();
        }

        @Override
        public String toString() {
            if (left == null && right == null) {
                return Integer.toString(root);
            }
            String l = (left == null) ? "-" : left.toString();
            String r = (right == null) ? "-" : right.toString();
            return root + "(" + l + "," + r + ")";
        }
    }
```

Listing 8.11 (Continued) *TripleTree.java: tripling every node by delegation, with a crashing wrong variant.*

```
    static Tree sample() {
        return new Tree(3,
                new Tree(4, new Tree(6), new Tree(7)),
                new Tree(5, new Tree(8), new Tree(9)));
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        Tree t = sample();
        System.out.println("before: " + t);
        t.triple();
        System.out.println("after : " + t);
        check("triple sample", "9(12(18,21),15(24,27))", t.toString());

        Tree lopsided = new Tree(1, new Tree(2), null);   // no right subtree
        lopsided.triple();
        check("triple lopsided", "3(6,-)", lopsided.toString());
        Tree leaf = new Tree(-4);
        leaf.triple();
        check("triple leaf", "-12", leaf.toString());

        Tree w = sample();
        try {
            w.tripleWrong();
        } catch (NullPointerException e) {
            System.out.println("tripleWrong crashed: NullPointerException");
            System.out.println("half-done tree   : " + w);
        }
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
before: 3(4(6,7),5(8,9))
after : 9(12(18,21),15(24,27))
PASS triple sample
PASS triple lopsided
PASS triple leaf
tripleWrong crashed: NullPointerException
half-done tree   : 9(12(18,7),5(8,9))
All 3 checks passed.
```

The Python solution. **Listing 8.12** follows the same plan.

Listing 8.12 `triple_tree.py`: tripling every node in Python.

```python
# triple_tree.py - Problem 8.2: multiply every node of a tree by three.
import sys
import unittest


class Tree:
    def __init__(self, root, left=None, right=None):
        self._root = root
        self._left = left
        self._right = right

    def triple(self):
        """Triple every node: subtrees first (by delegation), then my root."""
        if self._left is not None:
            self._left.triple()
        if self._right is not None:
            self._right.triple()
        self._root = 3 * self._root

    def triple_wrong(self):
        """Tempting but wrong: forgets that a subtree may be missing."""
        self._root = 3 * self._root
        self._left.triple_wrong()
        self._right.triple_wrong()

    def __str__(self):
        if self._left is None and self._right is None:
            return str(self._root)
        l = "-" if self._left is None else str(self._left)
        r = "-" if self._right is None else str(self._right)
        return f"{self._root}({l},{r})"


def sample():
    return Tree(3, Tree(4, Tree(6), Tree(7)), Tree(5, Tree(8), Tree(9)))


class TestTriple(unittest.TestCase):
    def test_sample(self):
        t = sample()
        t.triple()
        self.assertEqual("9(12(18,21),15(24,27))", str(t))
```

Listing 8.12 (Continued) *triple_tree.py: tripling every node in Python.*

```
    def test_lopsided(self):
        t = Tree(1, Tree(2))
        t.triple()
        self.assertEqual("3(6,-)", str(t))

    def test_wrong_crashes(self):
        with self.assertRaises(AttributeError):
            sample().triple_wrong()


if __name__ == "__main__":
    t = sample()
    print("before:", t)
    t.triple()
    print("after :", t)
    w = sample()
    try:
        w.triple_wrong()
    except AttributeError as e:
        print("triple_wrong crashed:", e)
        print("half-done tree      :", w)
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
before: 3(4(6,7),5(8,9))
after : 9(12(18,21),15(24,27))
triple_wrong crashed: 'NoneType' object has no attribute 'triple_wrong'
half-done tree      : 9(12(18,7),5(8,9))
test_lopsided (__main__.TestTriple.test_lopsided) ... ok
test_sample (__main__.TestTriple.test_sample) ... ok
test_wrong_crashes (__main__.TestTriple.test_wrong_crashes) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Line by line. The method has exactly the shape of `mirror()` in **Listing 8.1**: two guarded delegations, then the node's own job. The order of the three steps does not matter here, because tripling a node does not depend on its children; the hint says "subtrees first", and we follow it. The test `triple lopsided` builds a tree with a left subtree and no right subtree, the case that exposes a missing `null` check; `triple leaf` checks the smallest possible tree and a negative value.

A tempting wrong solution. `tripleWrong()` drops the two `if` tests, on the reasoning that "every node has two children". It does not: a leaf has none. The run shows what happens. The root becomes 9, the left child 12, the leaf 6 becomes 18, and then the leaf calls `left.tripleWrong()` on a `null` reference. Java throws `NullPointerException`; Python raises `AttributeError: 'NoneType' object has no attribute 'triple_wrong'`. Worse than the crash is what it leaves behind: `9(12(18,7),5(8,9))`, a tree that is partly tripled. An exception in the middle of a method that changes an object can leave the object in a state that no correct program would produce, which is one reason Chapter 5 recommended checking inputs *before* changing anything.

> **CHECK YOURSELF.** Write `void addToAll(int k)` that adds `k` to every node. What does `1(2,-)` become after `addToAll(10)`?
>
> *Answer.* The same shape with `root = root + k` as the node's own job: `11(12,-)`. The `null` guards stay exactly as in `triple()`.

8.4.3 Problem 8.3 — Printing the depth-first traversal

> **PROBLEM.** Include a recursive method `public void printDepthFirst()` in `Tree` that prints the depth-first traversal of the tree, choosing left branches first. For the tree of **Figure 8.4** (left) it should print `3 4 6 7 5 8 9`. Hint: print your root, then ask your left subtree to print its traversal, and finally ask your right subtree.

In our own words. A *traversal* visits every node exactly once. *Depth-first, left first, root before children* is also called *pre-order*: a node is printed before anything below it, and its entire left subtree is printed before any of its right subtree.

Design. **Table 8.4** traces the calls on the sample tree. Each row is one call; the indentation of the first column is the depth of the recursion.

Table 8.4 The seven calls of `printDepthFirst()` on the sample tree, in the order they happen. Reading the middle column from top to bottom gives the output.

Call	Prints	Then calls
node 3	3	node 4, then node 5
node 4	4	node 6, then node 7
node 6	6	nothing (leaf)
node 7	7	nothing (leaf)
node 5	5	node 8, then node 9
node 8	8	nothing (leaf)
node 9	9	nothing (leaf)

A method that *prints* is harder to test than a method that *returns*, because its result goes to the console instead of back to the caller. Chapter 6 showed the remedy: temporarily redirect the console into a buffer, run the method, restore the console, and compare the buffer's contents with the expected text.

The Java solution. **Listing 8.13** does exactly that in the helper `captured`.

Listing 8.13 `DepthFirstPrint.java`: pre-order printing, with a helper that captures console output for testing.

```java
// DepthFirstPrint.java - Problem 8.3: print the depth-first (pre-order) traversal.
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;

public class DepthFirstPrint {

    static final class Tree {
        private int root;
        private Tree left, right;

        Tree(int root, Tree left, Tree right) {
            this.root = root;
            this.left = left;
            this.right = right;
        }

        Tree(int root) {
            this(root, null, null);
        }

        /** Root first, then the whole left subtree, then the whole right subtree. */
        void printDepthFirst() {
            System.out.print(root + " ");
            if (left != null) {
                left.printDepthFirst();
            }
            if (right != null) {
                right.printDepthFirst();
            }
        }

        /** Tempting but wrong: the root is printed between its subtrees. */
        void printDepthFirstWrong() {
            if (left != null) {
                left.printDepthFirstWrong();
            }
            System.out.print(root + " ");
            if (right != null) {
                right.printDepthFirstWrong();
            }
        }
    }
```

Listing 8.13 (Continued) *DepthFirstPrint.java: pre-order printing, with a helper that captures console output for testing.*

```
    /** Runs an action and returns everything it printed (for testing). */
    static String captured(Runnable action) {
        PrintStream original = System.out;
        ByteArrayOutputStream buffer = new ByteArrayOutputStream();
        System.setOut(new PrintStream(buffer));
        try {
            action.run();
        } finally {
            System.setOut(original);        // always restore the real console
        }
        return buffer.toString();
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        Tree t = new Tree(3,
                new Tree(4, new Tree(6), new Tree(7)),
                new Tree(5, new Tree(8), new Tree(9)));
        t.printDepthFirst();
        System.out.println();
        t.printDepthFirstWrong();
        System.out.println(" <- wrong order");
        check("sample", "3 4 6 7 5 8 9 ", captured(t::printDepthFirst));
        Tree chain = new Tree(1, null, new Tree(2, new Tree(3), null));
        check("chain", "1 2 3 ", captured(chain::printDepthFirst));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
3 4 6 7 5 8 9
6 4 7 3 8 5 9  <- wrong order
PASS sample
PASS chain
All 2 checks passed.
```

The Python solution. In **Listing 8.14**, `contextlib.redirect_stdout` does the redirection and undoes it automatically at the end of the `with` block.

Listing 8.14 `depth_first_print.py`: pre-order printing in Python, tested by capturing standard output.

```
# depth_first_print.py - Problem 8.3: print the depth-first (pre-order) traversal.
import io
import sys
import unittest
from contextlib import redirect_stdout


class Tree:
    def __init__(self, root, left=None, right=None):
        self._root = root
        self._left = left
        self._right = right

    def print_depth_first(self):
        """Root first, then the whole left subtree, then the whole right subtree."""
        print(self._root, end=" ")
        if self._left is not None:
            self._left.print_depth_first()
        if self._right is not None:
            self._right.print_depth_first()

    def print_depth_first_wrong(self):
        """Tempting but wrong: the root is printed between its subtrees."""
        if self._left is not None:
            self._left.print_depth_first_wrong()
        print(self._root, end=" ")
        if self._right is not None:
            self._right.print_depth_first_wrong()


def captured(action):
    """Runs an action and returns everything it printed (for testing)."""
    buffer = io.StringIO()
    with redirect_stdout(buffer):       # restored automatically afterwards
        action()
    return buffer.getvalue()


def sample():
    return Tree(3, Tree(4, Tree(6), Tree(7)), Tree(5, Tree(8), Tree(9)))


class TestDepthFirst(unittest.TestCase):
    def test_sample(self):
        self.assertEqual("3 4 6 7 5 8 9 ", captured(sample().print_depth_first))
```

Listing 8.14 (Continued) *depth_first_print.py: pre-order printing in Python, tested by capturing standard output.*

```
    def test_chain(self):
        chain = Tree(1, None, Tree(2, Tree(3)))
        self.assertEqual("1 2 3 ", captured(chain.print_depth_first))


if __name__ == "__main__":
    sample().print_depth_first()
    print()
    sample().print_depth_first_wrong()
    print(" <- wrong order")
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
3 4 6 7 5 8 9
6 4 7 3 8 5 9  <- wrong order
test_chain (__main__.TestDepthFirst.test_chain) ... ok
test_sample (__main__.TestDepthFirst.test_sample) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Line by line. `printDepthFirst()` prints `root + " "` first, so every number is followed by one space, including the last; the expected strings in the checks therefore end in a space, `"3 4 6 7 5 8 9 "`. In `captured`, `System.setOut` swaps the console for a `PrintStream` that writes into a `ByteArrayOutputStream`. The `finally` block restores the original console *even if the action throws an exception*; without it, one failing test would leave every later message invisible. The parameter type `Runnable` is an interface from the Java library with a single method `run()`, and the *method reference* `t::printDepthFirst` is a compact way of writing an object that implements it by calling `t.printDepthFirst()`. The second test uses a "chain", a tree in which every node has one child, a shape that catches code assuming two children.

A tempting wrong solution. Moving the `print` between the two recursive calls looks harmless, but it changes the traversal. `printDepthFirstWrong()` prints `6 4 7 3 8 5 9`: every left subtree, then the node, then the right subtree. That is the *in-order* traversal, which is useful in its own right (Problem 8.7 uses it), but it is not the one asked for. The position of the node's own job relative to the delegations is the whole difference between pre-order, in-order and post-order.

> **CHECK YOURSELF.** What does a *post-order* traversal (left subtree, right subtree, then the node) print for the sample tree, and for the tree of **Figure 8.1**?
>
> *Answer.* Sample tree: `6 7 4 8 9 5 3`. **Figure 8.1**: `8 4 5 2 6 9 7 3 1`. In post-order the root is always printed last.

8.4.4 Problem 8.4 — A fresh copy of a tree

> **PROBLEM.** Include a recursive method `public Tree createFreshCopy()` in `Tree` that generates a fresh copy of a tree. After `Tree u = t.createFreshCopy(); t.mirror();`, the tree `u` must be exactly as `t` was before mirroring. Use one `new` statement per node; do not use Java's `clone` facility. Hint: ask your subtrees for fresh copies of themselves, then use them to build a fresh copy of yourself with `new Tree`.

In our own words. We need a *deep copy*: a second tree with the same shape and values, made of entirely new node objects, so that changing one tree cannot affect the other. The test given in the statement is precisely a test of independence: mirror the original, and the copy must not move.

Design. A fresh copy of a node is a new node holding the same root and *fresh copies* of both subtrees. A fresh copy of a missing subtree is a missing subtree. **Table 8.5** states it in the recipe's terms.

Table 8.5 The delegation recipe for a deep copy. Unlike `triple()`, the results of the delegated calls are the whole point.

Question	Answer for `createFreshCopy()`
Base case	A missing subtree copies to `null`
Delegation	`left.createFreshCopy()` and `right.createFreshCopy()`, and *keep the results*
My own job	`return new Tree(root, leftCopy, rightCopy)`
Returns	the new tree; `this` is unchanged

Figure 8.5 shows the difference between a deep copy and the tempting shallow one discussed below, in terms of objects and references.

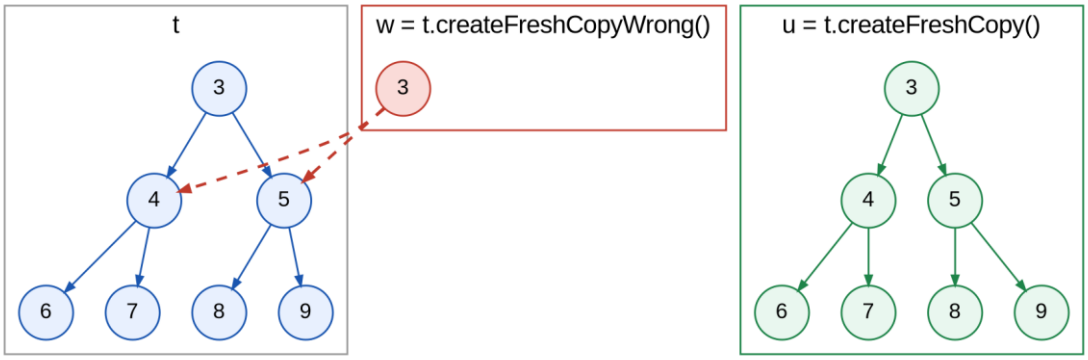


Figure 8.5 Left: the original `t`. Right: the fresh copy `u`, seven new objects. Middle: the wrong copy `w`, one new object whose references point back into `t`.

The Java solution. **Listing 8.15** also contains a method `sharesNodeWith` that answers "do these two trees have any node *object* in common?", using `==`, which in Java compares references, not values.

Listing 8.15 `FreshCopy.java`: a deep copy, a shallow wrong copy, and a test for shared node objects.

```java
// FreshCopy.java - Problem 8.4: a deep ("fresh") copy of a tree, one "new" per node.
public class FreshCopy {

    static final class Tree {
        private int root;
        private Tree left, right;

        Tree(int root, Tree left, Tree right) {
            this.root = root;
            this.left = left;
            this.right = right;
        }

        Tree(int root) {
            this(root, null, null);
        }

        /** Subtrees copy themselves and hand the copies back; I wrap them in a new node.
*/
        Tree createFreshCopy() {
            Tree leftCopy = (left == null) ? null : left.createFreshCopy();
            Tree rightCopy = (right == null) ? null : right.createFreshCopy();
            return new Tree(root, leftCopy, rightCopy);
        }

        /** Tempting but wrong: the recursive results are thrown away. */
        Tree createFreshCopyWrong() {
            if (left != null) {
                left.createFreshCopyWrong();      // result ignored!
            }
            if (right != null) {
                right.createFreshCopyWrong();     // result ignored!
            }
            return new Tree(root, left, right);   // shares my subtrees
        }

        void mirror() {
            if (left != null) {
                left.mirror();
            }
            if (right != null) {
                right.mirror();
            }
            Tree originalLeft = left;
            left = right;
            right = originalLeft;
        }
```

Listing 8.15 (Continued) *FreshCopy.java: a deep copy, a shallow wrong copy, and a test for shared node objects.*

```
        /** True if this tree and the other share at least one node object. */
        boolean sharesNodeWith(Tree other) {
            return other != null
                    && (contains(other)
                        || sharesNodeWith(other.left)
                        || sharesNodeWith(other.right));
        }

        /** True if node (the very same object) occurs somewhere in this tree. */
        private boolean contains(Tree node) {
            return this == node
                    || (left != null && left.contains(node))
                    || (right != null && right.contains(node));
        }

        @Override
        public String toString() {
            if (left == null && right == null) {
                return Integer.toString(root);
            }
            String l = (left == null) ? "-" : left.toString();
            String r = (right == null) ? "-" : right.toString();
            return root + "(" + l + "," + r + ")";
        }
    }

    static Tree sample() {
        return new Tree(3,
                new Tree(4, new Tree(6), new Tree(7)),
                new Tree(5, new Tree(8), new Tree(9)));
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }
```

Listing 8.15 (Continued) *FreshCopy.java: a deep copy, a shallow wrong copy, and a test for shared node objects.*

```
    public static void main(String[] args) {
        Tree t = sample();
        Tree u = t.createFreshCopy();
        t.mirror();
        System.out.println("t after mirror  = " + t);
        System.out.println("u (fresh copy)  = " + u);
        check("copy unchanged", "3(4(6,7),5(8,9))", u.toString());
        check("no shared nodes", false, t.sharesNodeWith(u));

        Tree t2 = sample();
        Tree w = t2.createFreshCopyWrong();
        t2.mirror();
        System.out.println("w (wrong copy)  = " + w);
        check("wrong copy shares nodes", true, t2.sharesNodeWith(w));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
t after mirror  = 3(5(9,8),4(7,6))
u (fresh copy)  = 3(4(6,7),5(8,9))
PASS copy unchanged
PASS no shared nodes
w (wrong copy)  = 3(4(7,6),5(9,8))
PASS wrong copy shares nodes
All 3 checks passed.
```

The Python solution. In **Listing 8.16**, the sharing test collects the identities `id(n)` of all nodes of one tree in a set and asks whether any node of the other tree has one of them.

Listing 8.16 `fresh_copy.py`: a deep copy in Python, with the same sharing test.

```
# fresh_copy.py - Problem 8.4: a deep ("fresh") copy of a tree, one Tree(...) per node.
import sys
import unittest


class Tree:
    def __init__(self, root, left=None, right=None):
        self._root = root
        self._left = left
        self._right = right

    def create_fresh_copy(self):
        """Subtrees copy themselves and hand the copies back; I wrap them in a new node."""
        left_copy = None if self._left is None else self._left.create_fresh_copy()
        right_copy = None if self._right is None else self._right.create_fresh_copy()
        return Tree(self._root, left_copy, right_copy)

    def create_fresh_copy_wrong(self):
        """Tempting but wrong: the recursive results are thrown away."""
        if self._left is not None:
            self._left.create_fresh_copy_wrong()        # result ignored!
        if self._right is not None:
            self._right.create_fresh_copy_wrong()       # result ignored!
        return Tree(self._root, self._left, self._right)  # shares my subtrees

    def mirror(self):
        if self._left is not None:
            self._left.mirror()
        if self._right is not None:
            self._right.mirror()
        self._left, self._right = self._right, self._left

    def nodes(self):
        """Every node object of this tree (used to detect sharing)."""
        result = [self]
        if self._left is not None:
            result += self._left.nodes()
        if self._right is not None:
            result += self._right.nodes()
        return result

    def shares_node_with(self, other):
        mine = {id(n) for n in self.nodes()}
        return any(id(n) in mine for n in other.nodes())
```

Listing 8.16 (Continued) *fresh_copy.py: a deep copy in Python, with the same sharing test.*

```
    def __str__(self):
        if self._left is None and self._right is None:
            return str(self._root)
        l = "-" if self._left is None else str(self._left)
        r = "-" if self._right is None else str(self._right)
        return f"{self._root}({l},{r})"


def sample():
    return Tree(3, Tree(4, Tree(6), Tree(7)), Tree(5, Tree(8), Tree(9)))


class TestFreshCopy(unittest.TestCase):
    def test_copy_survives_mirror(self):
        t = sample()
        u = t.create_fresh_copy()
        t.mirror()
        self.assertEqual("3(4(6,7),5(8,9))", str(u))
        self.assertFalse(t.shares_node_with(u))

    def test_wrong_copy_shares(self):
        t = sample()
        self.assertTrue(t.shares_node_with(t.create_fresh_copy_wrong()))


if __name__ == "__main__":
    t = sample()
    u = t.create_fresh_copy()
    w = t.create_fresh_copy_wrong()
    t.mirror()
    print("t after mirror  =", t)
    print("u (fresh copy)  =", u)
    print("w (wrong copy)  =", w)
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
t after mirror  = 3(5(9,8),4(7,6))
u (fresh copy)  = 3(4(6,7),5(8,9))
w (wrong copy)  = 3(4(7,6),5(9,8))
test_copy_survives_mirror (__main__.TestFreshCopy.test_copy_survives_mirror) ... ok
test_wrong_copy_shares (__main__.TestFreshCopy.test_wrong_copy_shares) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Line by line. `createFreshCopy()` stores the two delegated results in local variables, `leftCopy` and `rightCopy`, and passes them to the constructor. The conditional expression `(left == null) ? null : left.createFreshCopy()` is the base case and the delegation in one line. The number of `new` statements executed equals the number of nodes, seven for the sample tree, as the statement requires. The checks confirm both halves of the specification: `u` still prints the original shape after `t.mirror()`, and `t` and `u` share no node object.

A tempting wrong solution. `createFreshCopyWrong()` makes the recursive calls, which look like the hint, but throws their results away, and then builds `new Tree(root, left, right)` from the *original* subtrees. Only the root is new. The run shows the damage: after `t2.mirror()`, the wrong copy prints `3(4(7,6),5(9,8))`. Its root still has 4 on the left, because the new root object was not mirrored, but everything below has been mirrored, because it is the same objects as in `t2`. This is a *shallow copy*, and the half-right output makes it treacherous: a quick look says “the 4 is still on

the left, it worked". The sharing test says otherwise. Python's `copy.copy` makes shallow copies and `copy.deepcopy` deep ones; the problem forbids the library in order to make you see what "deep" means.

> **CHECK YOURSELF.** Write `Tree mirroredCopy()` that returns a *mirrored* fresh copy and leaves `this` untouched. What does it return for the sample tree?
>
> *Answer.* Swap the arguments: `return new Tree(root, rightCopy, leftCopy)`, where the copies are themselves `mirroredCopy()` of the subtrees. For the sample tree it returns `3(5(9,8),4(7,6))`, and `t` still prints `3(4(6,7),5(8,9))`.

8.4.5 Problem 8.5 — Saving the traversal into an array

> **PROBLEM.** Include a recursive method `public int saveDepthFirst(int a[], int whereToStart)` in `Tree` that stores the depth-first traversal of the tree into the array `a`, beginning at position `whereToStart`, and returns the last position used. For the tree with root 4, children 5 and 6, and grandchildren 7, 8, 9, 10, the fragment `int a[] = new int[13]; int end = t.saveDepthFirst(a, 4);` must fill the array as `? ? ? ? 4 5 7 8 6 9 10 ? ?` and set `end` to 10.

In our own words. This is Problem 8.3 again, except that instead of printing each value we write it into the next free cell of an array (Chapter 5), and the method must tell its caller where it stopped, so that the caller knows where to continue.

Design. The difficulty is bookkeeping. Each call writes its root at `whereToStart`; the left subtree must start one cell later; the right subtree must start one cell after the *last cell the left subtree used*, and nobody knows that number in advance. That is why the method returns it. **Table 8.6** traces the calls. The variable `last` always holds the last position written so far.

Table 8.6 The calls of `saveDepthFirst` on the sample tree starting at position 4. Each call returns the last position it wrote; the caller adds one to find the next free cell.

Call	Writes	Left call starts at	Left returns	Right call starts at	Returns
node 4, start 4	`a[4]=4`	5	7	8	10
node 5, start 5	`a[5]=5`	6	6	7	7
leaf 7, start 6	`a[6]=7`	—	—	—	6
leaf 8, start 7	`a[7]=8`	—	—	—	7
node 6, start 8	`a[8]=6`	9	9	10	10
leaf 9, start 9	`a[9]=9`	—	—	—	9
leaf 10, start 10	`a[10]=10`	—	—	—	10

The convention matters, so let us fix it in one sentence: *the method returns the last position written, and every caller starts the next piece at that position plus one.* If a tree has n nodes and the traversal starts at position s, the last position written is

$$end = s + n - 1, \tag{8.2}$$

which for the sample tree gives $4 + 7 - 1 = 10$, as required. Equation (8.2) is also a useful test: whatever the shape, only the node count and the start decide the returned value.

The Java solution. **Listing 8.17** implements the convention, together with a wrong variant.

Listing 8.17 `SaveDepthFirst.java`: writing the pre-order traversal into an array and returning the last position used.

```java
// SaveDepthFirst.java - Problem 8.5: store the depth-first traversal into an array.
import java.util.Arrays;

public class SaveDepthFirst {

    static final class Tree {
        private int root;
        private Tree left, right;

        Tree(int root, Tree left, Tree right) {
            this.root = root;
            this.left = left;
            this.right = right;
        }

        Tree(int root) {
            this(root, null, null);
        }

        /**
         * Writes my depth-first traversal into a, starting at whereToStart.
         * Returns the LAST position written.
         */
        int saveDepthFirst(int[] a, int whereToStart) {
            a[whereToStart] = root;
            int last = whereToStart;                    // so far, only my root
            if (left != null) {
                last = left.saveDepthFirst(a, last + 1);
            }
            if (right != null) {
                last = right.saveDepthFirst(a, last + 1);
            }
            return last;
        }

        /** Tempting but wrong: the right subtree starts ON the last cell, not after it. */
        int saveDepthFirstWrong(int[] a, int whereToStart) {
            a[whereToStart] = root;
            int last = whereToStart;
            if (left != null) {
                last = left.saveDepthFirstWrong(a, last + 1);
            }
            if (right != null) {
                last = right.saveDepthFirstWrong(a, last);
            }
            return last;
```

Listing 8.17 (Continued) *SaveDepthFirst.java: writing the pre-order traversal into an array and returning the last position used.*

```
        }
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        Tree t = new Tree(4,
                new Tree(5, new Tree(7), new Tree(8)),
                new Tree(6, new Tree(9), new Tree(10)));
        int[] a = new int[13];                  // Java fills a new int[] with 0
        int end = t.saveDepthFirst(a, 4);
        System.out.println("a   = " + Arrays.toString(a));
        System.out.println("end = " + end);
        check("array", "[0, 0, 0, 0, 4, 5, 7, 8, 6, 9, 10, 0, 0]", Arrays.toString(a));
        check("end", 10, end);

        int[] b = new int[3];
        check("single leaf", 0, new Tree(42).saveDepthFirst(b, 0));

        int[] c = new int[13];
        int wrongEnd = t.saveDepthFirstWrong(c, 4);
        System.out.println("wrong a = " + Arrays.toString(c) + ", end = " + wrongEnd);
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
a   = [0, 0, 0, 0, 4, 5, 7, 8, 6, 9, 10, 0, 0]
end = 10
PASS array
PASS end
PASS single leaf
wrong a = [0, 0, 0, 0, 4, 5, 6, 10, 0, 0, 0, 0, 0], end = 7
All 3 checks passed.
```

The Python solution. **Listing 8.18** uses a list of thirteen zeros in place of the Java array.

Listing 8.18 `save_depth_first.py`: the same traversal saved into a Python list.

```
# save_depth_first.py - Problem 8.5: store the depth-first traversal into an array.
import sys
import unittest


class Tree:
    def __init__(self, root, left=None, right=None):
        self._root = root
        self._left = left
        self._right = right

    def save_depth_first(self, a, where_to_start):
        """Writes my depth-first traversal into a from where_to_start.
        Returns the LAST position written."""
        a[where_to_start] = self._root
        last = where_to_start                       # so far, only my root
        if self._left is not None:
            last = self._left.save_depth_first(a, last + 1)
        if self._right is not None:
            last = self._right.save_depth_first(a, last + 1)
        return last

    def save_depth_first_wrong(self, a, where_to_start):
        """Tempting but wrong: the right subtree starts ON the last cell."""
        a[where_to_start] = self._root
        last = where_to_start
        if self._left is not None:
            last = self._left.save_depth_first_wrong(a, last + 1)
        if self._right is not None:
            last = self._right.save_depth_first_wrong(a, last)
        return last


def sample():
    return Tree(4, Tree(5, Tree(7), Tree(8)), Tree(6, Tree(9), Tree(10)))


class TestSave(unittest.TestCase):
    def test_sample(self):
        a = [0] * 13
        end = sample().save_depth_first(a, 4)
        self.assertEqual([0, 0, 0, 0, 4, 5, 7, 8, 6, 9, 10, 0, 0], a)
        self.assertEqual(10, end)
```

Listing 8.18 (Continued) *save_depth_first.py: the same traversal saved into a Python list.*

```
    def test_single_leaf(self):
        self.assertEqual(0, Tree(42).save_depth_first([0] * 3, 0))


if __name__ == "__main__":
    a = [0] * 13                    # thirteen zeros, like Java's new int[13]
    end = sample().save_depth_first(a, 4)
    print("a   =", a)
    print("end =", end)
    c = [0] * 13
    wrong_end = sample().save_depth_first_wrong(c, 4)
    print(f"wrong a = {c}, end = {wrong_end}")
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
a   = [0, 0, 0, 0, 4, 5, 7, 8, 6, 9, 10, 0, 0]
end = 10
wrong a = [0, 0, 0, 0, 4, 5, 6, 10, 0, 0, 0, 0, 0], end = 7
test_sample (__main__.TestSave.test_sample) ... ok
test_single_leaf (__main__.TestSave.test_single_leaf) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Line by line. The first line of the method writes the root. The local variable `last` starts equal to `whereToStart`, because so far only the root has been written. If there is a left subtree, it starts at `last + 1` and its answer replaces `last`; the same happens for the right subtree. The method returns `last`. Note what the unused cells contain. The statement writes them as `?`, but Java does not leave memory undefined: `new int[13]` fills a new array with zeros (Chapter 5), which is why the output shows `0` in positions 0 to 3, 11 and 12. The Python program imitates this with `[0] * 13`. The `single leaf` check confirms the smallest case of Equation (8.2): one node starting at 0 ends at 0.

A tempting wrong solution. The hint in the original statement speaks of the "next available position", and a programmer who half-follows it writes the right call as `right.saveDepthFirst(a, last)`, starting the right subtree *at* the last written cell instead of after it. The output shows the result: `[0, 0, 0, 0, 4, 5, 6, 10, 0, ...]` with `end = 7`. Leaf 8 overwrote leaf 7 in `a[6]`, node 6 overwrote leaf 8, and leaf 10 overwrote leaf 9; three values vanished without any error message. Mixing two conventions ("return the last used" and "return the next free") is the classic source of this off-by-one error. Choose one, write it in the documentation comment, and make every call follow it.

> **CHECK YOURSELF.** Rewrite the method with the other convention: it returns the *next free* position. What does it return for the sample tree starting at 4, and what are the recursive calls?
>
> *Answer.* It returns 11 ($s + n$). The node writes `a[next] = root`, sets `next = next + 1`, then `next = left.save(a, next)` and `next = right.save(a, next)`, with no `+ 1` anywhere in the calls, and returns `next`.

8.4.6 Problem 8.6 — Finding a value in a binary search tree

> **PROBLEM.** Recall that in a *binary search tree* (BST) all nodes in the left branch are smaller than the root, all nodes in the right branch are greater, and both branches are themselves BSTs. Include a recursive method `public Tree find(int n)` in `Tree` that, assuming the tree is a BST, returns a reference to the subtree whose root equals `n`, or `null` if there is none. It must be efficient: it should follow only the single path down from the root, not search the whole tree. Hint: check your root; if the value is not there, decide whether to ask left or right, but not both.

In our own words. The ordering rule means that one comparison at a node tells us which half cannot contain `n`. We then continue in the other half only. **Figure 8.6** shows the test tree, with the path followed by `find(8)`: the root 5, then 8, found, after two visits.

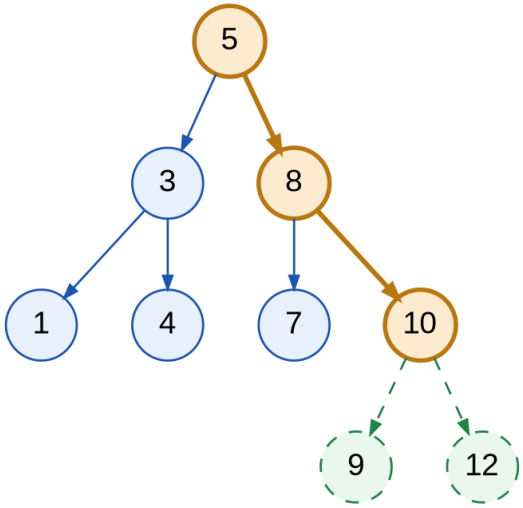


Figure 8.6 The binary search tree used in Problems 8.6 and 8.7. The thick path is followed by `find(8)` and `find(10)`; the dashed leaves 9 and 12 are added by `insert(9)` and `insert(12)`.

Design. **Table 8.7** lists the four possible situations at a node.

Table 8.7 The cases of `find(n)` in a BST. Exactly one of the two subtrees is ever asked.

Situation at a node	Action
`n == root`	found: `return this`
`n < root`, left subtree exists	`return left.find(n)`
`n > root`, right subtree exists	`return right.find(n)`
the needed subtree is missing	not in the tree: `return null`

Each call either stops or moves one level down, so the number of visited nodes is at most the *height* h of the tree, the number of nodes on the longest path from the root to a leaf. A balanced tree with n nodes has height about $\log_2 n$, so

$$\text{visits(find)} \le h, \qquad h \approx \log_2 n \text{ for a balanced tree,} \tag{8.3}$$

which for a million nodes is about 20 visits instead of a million.

The Java solution. **Listing 8.19** counts the visits in a `static` field so that the output can show the single path.

Listing 8.19 `BstFind.java`: searching a BST along one path, with a visit counter and a crashing wrong variant.

```java
// BstFind.java - Problem 8.6: find a value in a binary search tree along ONE path.
public class BstFind {

    static final class Tree {
        private int root;
        private Tree left, right;
        static int visits = 0;                // how many nodes find() looked at

        Tree(int root, Tree left, Tree right) {
            this.root = root;
            this.left = left;
            this.right = right;
        }

        Tree(int root) {
            this(root, null, null);
        }

        /** Returns the subtree whose root is n, or null if n is not in this BST. */
        Tree find(int n) {
            visits++;
            if (n == root) {
                return this;
            } else if (n < root) {
                return (left == null) ? null : left.find(n);
            } else {
                return (right == null) ? null : right.find(n);
            }
        }

        /** Tempting but wrong: walks into a missing subtree. */
        Tree findWrong(int n) {
            if (n == root) {
                return this;
            } else if (n < root) {
                return left.findWrong(n);
            } else {
                return right.findWrong(n);
            }
        }
```

Listing 8.19 (Continued) *BstFind.java: searching a BST along one path, with a visit counter and a crashing wrong variant.*

```
        @Override
        public String toString() {
            if (left == null && right == null) {
                return Integer.toString(root);
            }
            String l = (left == null) ? "-" : left.toString();
            String r = (right == null) ? "-" : right.toString();
            return root + "(" + l + "," + r + ")";
        }
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        Tree t = new Tree(5,
                new Tree(3, new Tree(1), new Tree(4)),
                new Tree(8, new Tree(7), new Tree(10)));
        for (int n : new int[] {8, 4, 5, 6}) {
            Tree.visits = 0;
            Tree found = t.find(n);
            System.out.println("find(" + n + ") = " + found + ", nodes visited: " +
Tree.visits);
        }
        check("find 8", "8(7,10)", String.valueOf(t.find(8)));
        check("find 1", "1", String.valueOf(t.find(1)));
        check("find 6 is null", null, t.find(6));
        check("find 11 is null", null, t.find(11));
        try {
            t.findWrong(6);
        } catch (NullPointerException e) {
            System.out.println("findWrong(6) crashed: NullPointerException");
        }
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
find(8) = 8(7,10), nodes visited: 2
find(4) = 4, nodes visited: 3
find(5) = 5(3(1,4),8(7,10)), nodes visited: 1
find(6) = null, nodes visited: 3
PASS find 8
PASS find 1
PASS find 6 is null
PASS find 11 is null
findWrong(6) crashed: NullPointerException
All 4 checks passed.
```

The Python solution. **Listing 8.20** uses a *class attribute* `Tree.visits`, shared by all nodes, for the counter.

Listing 8.20 `bst_find.py`: the same search in Python.

```
# bst_find.py - Problem 8.6: find a value in a binary search tree along ONE path.
import sys
import unittest


class Tree:
    visits = 0                      # how many nodes find() looked at

    def __init__(self, root, left=None, right=None):
        self._root = root
        self._left = left
        self._right = right

    def find(self, n):
        """Returns the subtree whose root is n, or None if n is not in this BST."""
        Tree.visits += 1
        if n == self._root:
            return self
        elif n < self._root:
            return None if self._left is None else self._left.find(n)
        else:
            return None if self._right is None else self._right.find(n)

    def find_wrong(self, n):
        """Tempting but wrong: walks into a missing subtree."""
        if n == self._root:
            return self
        elif n < self._root:
            return self._left.find_wrong(n)
        else:
            return self._right.find_wrong(n)

    def __str__(self):
        if self._left is None and self._right is None:
            return str(self._root)
        l = "-" if self._left is None else str(self._left)
        r = "-" if self._right is None else str(self._right)
        return f"{self._root}({l},{r})"


def sample():
    return Tree(5, Tree(3, Tree(1), Tree(4)), Tree(8, Tree(7), Tree(10)))
```

Listing 8.20 (Continued) *bst_find.py: the same search in Python.*

```
class TestFind(unittest.TestCase):
    def test_present(self):
        self.assertEqual("8(7,10)", str(sample().find(8)))
        self.assertEqual("1", str(sample().find(1)))

    def test_absent(self):
        self.assertIsNone(sample().find(6))
        self.assertIsNone(sample().find(11))


if __name__ == "__main__":
    t = sample()
    for n in [8, 4, 5, 6]:
        Tree.visits = 0
        found = t.find(n)
        print(f"find({n}) = {found}, nodes visited: {Tree.visits}")
    try:
        t.find_wrong(6)
    except AttributeError as e:
        print("find_wrong(6) crashed:", e)
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
find(8) = 8(7,10), nodes visited: 2
find(4) = 4, nodes visited: 3
find(5) = 5(3(1,4),8(7,10)), nodes visited: 1
find(6) = None, nodes visited: 3
find_wrong(6) crashed: 'NoneType' object has no attribute 'find_wrong'
test_absent (__main__.TestFind.test_absent) ... ok
test_present (__main__.TestFind.test_present) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Line by line. The output confirms Equation (8.3): `find(5)` stops at the root after one visit, `find(8)` needs two, `find(4)` three (5, 3, 4), and the unsuccessful `find(6)` also three (5, 8, 7, and 7 has no left subtree). The height of this tree is 3, and no search ever visits more nodes. The method returns the *subtree*, not only a yes-or-no answer, so the caller can continue working from there; printing it shows `8(7,10)`. The counter `visits` is `static` because it belongs to the class, not to any one node; it is a testing aid, and a production class would not keep it.

A tempting wrong solution. `findWrong` omits the checks for a missing subtree. For values in the tree it works, which makes it easy to trust. For `find(6)` it goes 5, 8, 7, then calls `left.findWrong(6)` on 7's missing left subtree, and crashes with `NullPointerException` (Python: `AttributeError`). A second tempting mistake, not shown, searches *both* subtrees "to be safe": it always gives the right answer but visits every node, which is exactly what the statement forbids. A test that counts visits, like ours, is the only kind of test that catches it.

> **CHECK YOURSELF.** In the tree of **Figure 8.6** (without 9 and 12), which nodes does `find(11)` visit, and what does it return? And `find(1)`?
>
> *Answer.* `find(11)`: 5, 8, 10, then 10 has no right subtree, so it returns `null` after 3 visits. `find(1)`: 5, 3, 1, found after 3 visits; it returns the leaf `1`.

8.4.7 Problem 8.7 — Inserting a value into a binary search tree

> **PROBLEM.** Using the BST of Problem 8.6, include a recursive method `public Tree insert(int n)` in `Tree` that inserts a new node in the correct position, which will be a leaf.

> Return the new tree if you succeed, and the current tree if the element is already there and cannot be inserted. Hint: modify your solution to Problem 8.6. If you find the element, it cannot be inserted. If not, you reach a node whose needed subtree is missing, and the new element becomes that child.

In our own words. Walk down exactly as `find` does. If we meet `n`, stop and change nothing. If the subtree we need is missing, create the leaf *and attach it* there. The ordering rule is preserved automatically, because the new leaf sits exactly where `find` would look for it.

Design. **Table 8.8** compares `find` and `insert` case by case; only the last row differs.

Table 8.8 `insert` is `find` with a different last case: where the search would give up, the new leaf is attached.

Situation	`find(n)`	`insert(n)`
`n == root`	return `this`	nothing to do; return `this`
`n < root`, left exists	`left.find(n)`	`left.insert(n)`
`n > root`, right exists	`right.find(n)`	`right.insert(n)`
needed subtree missing	return `null`	`left = new Tree(n)` or `right = new Tree(n)`

The Java solution. **Listing 8.21** also adds an in-order printer (left, node, right), which for a correct BST must list the values in increasing order.

Listing 8.21 `BstInsert.java`: inserting into a BST by attaching a new leaf, with an in-order check.

```java
// BstInsert.java - Problem 8.7: insert a value into a binary search tree as a new leaf.
public class BstInsert {

    static final class Tree {
        private int root;
        private Tree left, right;

        Tree(int root, Tree left, Tree right) {
            this.root = root;
            this.left = left;
            this.right = right;
        }

        Tree(int root) {
            this(root, null, null);
        }

        /**
         * Inserts n as a new leaf in the correct place and returns this tree.
         * If n is already present, nothing changes and this tree is returned.
         */
        Tree insert(int n) {
            if (n < root) {
                if (left == null) {
                    left = new Tree(n);          // attach the new leaf here
                } else {
                    left.insert(n);
                }
            } else if (n > root) {
                if (right == null) {
                    right = new Tree(n);
                } else {
                    right.insert(n);
                }
            }
            return this;                         // n == root: already there
        }
```

Listing 8.21 (Continued) *BstInsert.java: inserting into a BST by attaching a new leaf, with an in-order check.*

```
        /** Tempting but wrong: creates the leaf but never attaches it. */
        Tree insertWrong(int n) {
            if (n < root) {
                return (left == null) ? new Tree(n) : left.insertWrong(n);
            } else if (n > root) {
                return (right == null) ? new Tree(n) : right.insertWrong(n);
            }
            return this;
        }

        /** In-order: left, root, right. For a BST this is increasing order. */
        String inOrder() {
            String l = (left == null) ? "" : left.inOrder();
            String r = (right == null) ? "" : right.inOrder();
            return l + root + " " + r;
        }

        @Override
        public String toString() {
            if (left == null && right == null) {
                return Integer.toString(root);
            }
            String l = (left == null) ? "-" : left.toString();
            String r = (right == null) ? "-" : right.toString();
            return root + "(" + l + "," + r + ")";
        }
    }

    static Tree sample() {
        return new Tree(5,
                new Tree(3, new Tree(1), new Tree(4)),
                new Tree(8, new Tree(7), new Tree(10)));
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }
```

Listing 8.21 (Continued) *BstInsert.java: inserting into a BST by attaching a new leaf, with an in-order check.*

```
    public static void main(String[] args) {
        Tree t = sample();
        t = t.insert(9);
        System.out.println("after insert(9) : " + t);
        t = t.insert(12);
        System.out.println("after insert(12): " + t);
        Tree before = t;
        t = t.insert(8);
        System.out.println("after insert(8) : " + t);
        check("shape", "5(3(1,4),8(7,10(9,12)))", t.toString());
        check("in-order is sorted", "1 3 4 5 7 8 9 10 12 ", t.inOrder());
        check("duplicate returns same object", true, before == t);

        Tree w = sample();
        w = w.insertWrong(9);
        System.out.println("wrong insert(9) : " + w);
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
after insert(9) : 5(3(1,4),8(7,10(9,-)))
after insert(12): 5(3(1,4),8(7,10(9,12)))
after insert(8) : 5(3(1,4),8(7,10(9,12)))
PASS shape
PASS in-order is sorted
PASS duplicate returns same object
wrong insert(9) : 9
All 3 checks passed.
```

The Python solution. **Listing 8.22** is line-for-line the same; `assertIs` checks object identity, the Python counterpart of Java's `==` on references.

Listing 8.22 `bst_insert.py`: BST insertion in Python.

```
# bst_insert.py - Problem 8.7: insert a value into a binary search tree as a new leaf.
import sys
import unittest


class Tree:
    def __init__(self, root, left=None, right=None):
        self._root = root
        self._left = left
        self._right = right

    def insert(self, n):
        """Inserts n as a new leaf in the correct place and returns this tree.
        If n is already present, nothing changes and this tree is returned."""
        if n < self._root:
            if self._left is None:
                self._left = Tree(n)             # attach the new leaf here
            else:
                self._left.insert(n)
        elif n > self._root:
            if self._right is None:
                self._right = Tree(n)
            else:
                self._right.insert(n)
        return self                              # n == root: already there

    def insert_wrong(self, n):
        """Tempting but wrong: creates the leaf but never attaches it."""
        if n < self._root:
            return Tree(n) if self._left is None else self._left.insert_wrong(n)
        elif n > self._root:
            return Tree(n) if self._right is None else self._right.insert_wrong(n)
        return self

    def in_order(self):
        """In-order: left, root, right. For a BST this is increasing order."""
        l = "" if self._left is None else self._left.in_order()
        r = "" if self._right is None else self._right.in_order()
        return l + f"{self._root} " + r
```

Listing 8.22 (Continued) *bst_insert.py: BST insertion in Python.*

```
    def __str__(self):
        if self._left is None and self._right is None:
            return str(self._root)
        l = "-" if self._left is None else str(self._left)
        r = "-" if self._right is None else str(self._right)
        return f"{self._root}({l},{r})"


def sample():
    return Tree(5, Tree(3, Tree(1), Tree(4)), Tree(8, Tree(7), Tree(10)))


class TestInsert(unittest.TestCase):
    def test_inserts(self):
        t = sample().insert(9).insert(12)
        self.assertEqual("5(3(1,4),8(7,10(9,12)))", str(t))
        self.assertEqual("1 3 4 5 7 8 9 10 12 ", t.in_order())

    def test_duplicate_returns_same_object(self):
        t = sample()
        self.assertIs(t, t.insert(8))
        self.assertEqual("5(3(1,4),8(7,10))", str(t))


if __name__ == "__main__":
    t = sample()
    t = t.insert(9)
    print("after insert(9) :", t)
    t = t.insert(12)
    print("after insert(12):", t)
    t = t.insert(8)
    print("after insert(8) :", t)
    w = sample().insert_wrong(9)
    print("wrong insert(9) :", w)
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
after insert(9) : 5(3(1,4),8(7,10(9,-)))
after insert(12): 5(3(1,4),8(7,10(9,12)))
after insert(8) : 5(3(1,4),8(7,10(9,12)))
wrong insert(9) : 9
test_duplicate_returns_same_object (__main__.TestInsert.test_duplicate_returns_same_object)
... ok
test_inserts (__main__.TestInsert.test_inserts) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Line by line. `insert(9)` walks 5, 8, 10; 9 is smaller than 10 and 10 has no left subtree, so `left = new Tree(9)` runs inside node 10. `insert(12)` walks the same path and attaches 12 on the right of 10. `insert(8)` stops at node 8 and changes nothing. In every case the method returns `this`, the root of the whole tree, so the calling pattern `t = t.insert(9)` from the statement works. Three checks cover the specification: the exact shape, the in-order listing `1 3 4 5 7 8 9 10 12` (which proves the BST rule still holds everywhere), and the fact that inserting a duplicate returns the very same object (`before == t`).

A tempting wrong solution. `insertWrong` creates the leaf with `new Tree(n)` and *returns* it, but never stores it in `left` or `right`. The new node is attached to nothing. Worse, because the method returns the new leaf instead of the root, the statement `w = w.insertWrong(9)` replaces the

caller's whole tree with the single node 9, as the output line `wrong insert(9) : 9` shows. Creating an object is not the same as connecting it; an object that no field refers to is simply garbage, which the Java garbage collector (or Python's) reclaims.

The statement's phrase "return the new tree" allows a second correct design, in which `insert` changes no existing node and instead builds new nodes along the search path, sharing the untouched subtrees with the old tree. That *persistent* version is Exercise 8.11.

> **CHECK YOURSELF.** Starting from the tree of **Figure 8.6** without 9 and 12, insert 6 and then 2. Where does each go, and what does `toString()` print afterwards?
>
> *Answer.* 6: path 5, 8, 7; 6 < 7 and 7 has no left child, so 6 becomes the left child of 7. 2: path 5, 3, 1; 2 > 1, so 2 becomes the right child of 1. Result: `5(3(1(-,2),4),8(7(6,-),10))`.

8.5 Problems on Recursive Lists

The eight problems of this section are static methods that take lists built by `List` (**Listing 8.3**) and, where a list is the answer, return a *new* list; no list is ever changed. Each Java program is compiled together with `List.java` and `ListException.java` (`javac *.java` in the chapter's folder does it all at once), and each Python program imports `cons_list`. The statement asks that only the operations of `List` and `ListOps` be used, so there are no loops and no Java or Python library lists: every problem is solved with `empty()`, `head()`, `tail()`, `nil()`, `cons()` and recursion.

Every list method below has the same skeleton, which is the delegation recipe of Section 8.2 specialized to lists:

$$f(a) = \begin{cases} \text{answer for the empty list} & \text{if } a = \text{nil}, \\ \text{combine}(\text{head}(a),\ f(\text{tail}(a))) & \text{otherwise.} \end{cases} \tag{8.4}$$

Solving a problem means filling in the two blanks of Equation (8.4). Because each call removes one element, a list of n elements causes $n + 1$ calls, and the running time is proportional to n whenever "combine" takes constant time.

8.5.1 Problem 8.8 — Negating a list

> **PROBLEM.** Include a recursive method `static List negateAll(List a)` that returns a new list with all the elements of `a` with their sign negated: positive integers become negative and negative integers become positive. Example: `[2, -5, 8, 0]` gives `[-2, 5, -8, 0]`.

In our own words. Build a list of the same length whose i-th element is minus the i-th element of `a`. The original list must stay as it was, which the immutable `List` guarantees anyway.

Design. **Table 8.9** fills the blanks of Equation (8.4).

Table 8.9 Design of `negateAll`: a *map* over the list, one output element per input element.

Part	Answer for `negateAll`
Empty list	`nil()`
Combine	`cons(-head, negateAll(tail))`
Edge cases to test	empty list; applying it twice gives the original

Listing 8.23 `NegateAll.java`: negating every element recursively.

```java
// NegateAll.java - Problem 8.8: a new list with every sign flipped.
public class NegateAll {

    /** [2, -5, 8, 0] becomes [-2, 5, -8, 0]; the list a itself is untouched. */
    static List negateAll(List a) {
        if (a.empty()) {
            return List.nil();                          // base case
        }
        return List.cons(-a.head(), negateAll(a.tail()));  // recursive case
    }

    /** Tempting but wrong: negates the head and forgets the rest. */
    static List negateAllWrong(List a) {
        if (a.empty()) {
            return List.nil();
        }
        return List.cons(-a.head(), a.tail());
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        List a = List.of(2, -5, 8, 0);
        System.out.println("a                = " + a);
        System.out.println("negateAll(a)     = " + negateAll(a));
        System.out.println("negateAllWrong(a) = " + negateAllWrong(a));
        check("sample", "[-2, 5, -8, 0]", negateAll(a).toString());
        check("a untouched", "[2, -5, 8, 0]", a.toString());
        check("empty", "[]", negateAll(List.nil()).toString());
        check("twice is identity", a.toString(), negateAll(negateAll(a)).toString());
        System.out.println("negate MIN_VALUE  = " + negateAll(List.of(Integer.MIN_VALUE)));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
a                = [2, -5, 8, 0]
negateAll(a)     = [-2, 5, -8, 0]
negateAllWrong(a) = [-2, -5, 8, 0]
PASS sample
PASS a untouched
PASS empty
PASS twice is identity
negate MIN_VALUE  = [-2147483648]
All 4 checks passed.
```

Listing 8.24 `negate_all.py`: the same function in Python.

```
# negate_all.py - Problem 8.8: a new list with every sign flipped.
import sys
import unittest
from cons_list import List


def negate_all(a):
    """[2, -5, 8, 0] becomes [-2, 5, -8, 0]; the list a itself is untouched."""
    if a.empty():
        return List.nil()                              # base case
    return List.cons(-a.head(), negate_all(a.tail()))   # recursive case


def negate_all_wrong(a):
    """Tempting but wrong: negates the head and forgets the rest."""
    if a.empty():
        return List.nil()
    return List.cons(-a.head(), a.tail())


class TestNegateAll(unittest.TestCase):
    def test_sample(self):
        a = List.of(2, -5, 8, 0)
        self.assertEqual("[-2, 5, -8, 0]", str(negate_all(a)))
        self.assertEqual("[2, -5, 8, 0]", str(a))

    def test_empty(self):
        self.assertEqual("[]", str(negate_all(List.nil())))

    def test_twice_is_identity(self):
        a = List.of(2, -5, 8, 0)
        self.assertEqual(str(a), str(negate_all(negate_all(a))))


if __name__ == "__main__":
    a = List.of(2, -5, 8, 0)
    print("a                  =", a)
    print("negate_all(a)      =", negate_all(a))
    print("negate_all_wrong(a) =", negate_all_wrong(a))
    print("negate -2147483648  =", negate_all(List.of(-2147483648)))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
a                  = [2, -5, 8, 0]
negate_all(a)      = [-2, 5, -8, 0]
negate_all_wrong(a) = [-2, -5, 8, 0]
negate -2147483648  = [2147483648]
test_empty (__main__.TestNegateAll.test_empty) ... ok
test_sample (__main__.TestNegateAll.test_sample) ... ok
test_twice_is_identity (__main__.TestNegateAll.test_twice_is_identity) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Line by line. **Listing 8.23** and **Listing 8.24** are three lines each once the comments are removed: the base case, and one `cons` whose head is the negated head and whose tail is the recursive answer for the tail. The check `twice is identity` is a *property test*: for any list, negating twice must give the original, and it is a good habit to test such laws in addition to single examples. The last demonstration line holds a surprise for Java programmers. The smallest `int`, $-2^{31} = -2147483648$, has no positive partner in 32 bits (the largest `int` is $2^{31} - 1$), so `-a.head()` *overflows* and gives back -2147483648 unchanged. Python integers have no fixed size, so Python

prints `[2147483648]`. The statement does not mention this case, but a careful programmer notices it and documents it.

A tempting wrong solution. `negateAllWrong` negates the head and then attaches the original tail, `cons(-a.head(), a.tail())`, forgetting to recurse. It returns `[-2, -5, 8, 0]`: only the first element changed. A test with a one-element list would pass; the test on the four-element example fails.

> **CHECK YOURSELF.** Write `static List squares(List a)`. What does it return for `[3, -1, 0]`?
>
> *Answer.* The same shape with `cons(a.head() * a.head(), squares(a.tail()))`; it returns `[9, 1, 0]`.

8.5.2 Problem 8.9 — Finding the position of an element

> **PROBLEM.** Include a recursive method `static int find(int x, List a)` that returns the position of the first occurrence of `x` in `a`, counting positions from 0. If `x` does not occur, return `-1`. Examples: `x = 3`, `a = [7, 5, 3, 8]` gives 2; `x = 2` gives `-1`.

In our own words. If the head is `x`, the answer is 0. Otherwise the answer is one more than the position of `x` in the tail, *provided* `x` is in the tail at all.

Design. The subtle point is that the recursive answer can be a real position or the signal `-1`, and the two must be treated differently (**Table 8.10**).

Table 8.10 Design of `find`: a position found in the tail is shifted by one; the "not found" signal is not.

Situation	Answer
`a` is empty	`-1`
`head(a) == x`	`0`
`find(x, tail)` returns `-1`	`-1` (pass the signal on unchanged)
`find(x, tail)` returns $p \geq 0$	$p + 1$

Listing 8.25 (Java) and **Listing 8.26** (Python) treat the signal exactly as the table says; the tests are built into each program.

Listing 8.25 `ListFind.java`: the position of the first occurrence, or `-1`.

```
// ListFind.java - Problem 8.9: position of the first occurrence of x, or -1.
public class ListFind {

    /** Positions count 0, 1, 2, ...; -1 means "x does not occur in a". */
    static int find(int x, List a) {
        if (a.empty()) {
            return -1;                          // nothing left to search
        }
        if (a.head() == x) {
            return 0;                           // found right here
        }
        int inTail = find(x, a.tail());         // ask the rest of the list
        return (inTail == -1) ? -1 : inTail + 1;
    }

    /** Tempting but wrong: adds 1 even to the "not found" answer. */
    static int findWrong(int x, List a) {
        if (a.empty()) {
            return -1;
        }
        if (a.head() == x) {
            return 0;
        }
        return 1 + findWrong(x, a.tail());
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }
```

Listing 8.25 (Continued) *ListFind.java: the position of the first occurrence, or -1.*

```
    public static void main(String[] args) {
        List a = List.of(7, 5, 3, 8);
        System.out.println("a = " + a);
        System.out.println("find(3, a)      = " + find(3, a));
        System.out.println("find(2, a)      = " + find(2, a));
        System.out.println("findWrong(2, a) = " + findWrong(2, a));
        check("present", 2, find(3, a));
        check("absent", -1, find(2, a));
        check("first element", 0, find(7, a));
        check("last element", 3, find(8, a));
        check("first of duplicates", 1, find(5, List.of(9, 5, 5)));
        check("empty list", -1, find(1, List.nil()));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
a = [7, 5, 3, 8]
find(3, a)      = 2
find(2, a)      = -1
findWrong(2, a) = 3
PASS present
PASS absent
PASS first element
PASS last element
PASS first of duplicates
PASS empty list
All 6 checks passed.
```

Listing 8.26 `list_find.py`: the same search in Python.

```
# list_find.py - Problem 8.9: position of the first occurrence of x, or -1.
import sys
import unittest
from cons_list import List


def find(x, a):
    """Positions count 0, 1, 2, ...; -1 means "x does not occur in a"."""
    if a.empty():
        return -1                          # nothing left to search
    if a.head() == x:
        return 0                           # found right here
    in_tail = find(x, a.tail())            # ask the rest of the list
    return -1 if in_tail == -1 else in_tail + 1


def find_wrong(x, a):
    """Tempting but wrong: adds 1 even to the "not found" answer."""
    if a.empty():
        return -1
    if a.head() == x:
        return 0
    return 1 + find_wrong(x, a.tail())


class TestFind(unittest.TestCase):
    def test_present_and_absent(self):
        a = List.of(7, 5, 3, 8)
        self.assertEqual(2, find(3, a))
        self.assertEqual(-1, find(2, a))

    def test_ends(self):
        a = List.of(7, 5, 3, 8)
        self.assertEqual(0, find(7, a))
        self.assertEqual(3, find(8, a))

    def test_first_of_duplicates_and_empty(self):
        self.assertEqual(1, find(5, List.of(9, 5, 5)))
        self.assertEqual(-1, find(1, List.nil()))
```

Listing 8.26 (Continued) *list_find.py: the same search in Python.*

```
if __name__ == "__main__":
    a = List.of(7, 5, 3, 8)
    print("a =", a)
    print("find(3, a)       =", find(3, a))
    print("find(2, a)       =", find(2, a))
    print("find_wrong(2, a) =", find_wrong(2, a))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
a = [7, 5, 3, 8]
find(3, a)       = 2
find(2, a)       = -1
find_wrong(2, a) = 3
test_ends (__main__.TestFind.test_ends) ... ok
test_first_of_duplicates_and_empty (__main__.TestFind.test_first_of_duplicates_and_empty)
... ok
test_present_and_absent (__main__.TestFind.test_present_and_absent) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Line by line. For `find(3, [7, 5, 3, 8])` the calls go down `[7,5,3,8]`, `[5,3,8]`, `[3,8]`; the third finds 3 at its head and returns 0; the second returns $0 + 1 = 1$; the first returns $1 + 1 = 2$. Because the method stops at the first match, `find(5, [9, 5, 5])` returns 1, not 2, which the check `first of duplicates` confirms.

A tempting wrong solution. `findWrong` returns `1 + findWrong(x, a.tail())` without looking at what came back. When `x` is absent, the `-1` from the empty list is increased once per element: for four elements, $-1 + 4 = 3$. The output `findWrong(2, a) = 3` is the worst kind of wrong answer, because 3 is a valid position in the list (it holds 8). A caller would happily use it. Special "signal" values such as `-1` must be passed through unchanged; Java's `Optional` and Python's `None` exist partly to make that mistake harder.

> **CHECK YOURSELF.** Write `static int findLast(int x, List a)`, the position of the *last* occurrence. What does it return for `x = 5`, `a = [9, 5, 5]`?
>
> *Answer.* Ask the tail first: if `findLast(x, tail)` is $p \geq 0$, return $p + 1$; otherwise return 0 if the head is `x`, else `-1`. It returns 2.

8.5.3 Problem 8.10 — Are all elements positive?

> **PROBLEM.** Include a recursive method `static boolean allPositive(List a)` that returns whether all the elements of `a` are positive, where *positive* here means ≥ 0.

In our own words. Note the unusual definition: zero counts as "positive" in this problem. (In mathematics 0 is neither positive nor negative, and a clearer method name would be `allNonNegative`; but we follow the statement and document the choice.) The question is "is there any element below zero?", answered with *no*.

Design. **Table 8.11** shows the design. The empty list is the interesting case: it has no element that breaks the rule, so the answer is `true`. Logicians call this *vacuous truth*, and it is also the only choice that makes the recursion work, because `allPositive([x])` must equal `x >= 0 && allPositive([])`.

Table 8.11 Design of `allPositive`: a logical AND over all elements, with `true` as the neutral starting value.

Part	Answer for `allPositive`
Empty list	`true` (no counterexample exists)
Combine	`head >= 0 && allPositive(tail)`
Edge cases to test	`[]`, `[0]`, a negative element in the *last* position

Listing 8.27 (Java) and **Listing 8.28** (Python) follow the table; the tests are built into each program.

Listing 8.27 `AllPositive.java`: checking that no element is below zero.

```
// AllPositive.java - Problem 8.10: are all elements positive (here: >= 0)?
public class AllPositive {

    /** True when every element is >= 0; true for the empty list. */
    static boolean allPositive(List a) {
        if (a.empty()) {
            return true;                      // no element can break the rule
        }
        return a.head() >= 0 && allPositive(a.tail());
    }

    /** Tempting but wrong: stops one element too early. */
    static boolean allPositiveWrong(List a) {
        if (a.empty()) {
            return true;
        } else if (a.tail().empty()) {
            return true;                      // the last head is never checked!
        } else if (a.head() >= 0) {
            return allPositiveWrong(a.tail());
        }
        return false;
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        System.out.println("allPositive([7, 5, 3, 8])  = " + allPositive(List.of(7, 5, 3,
8)));
        System.out.println("allPositive([2, -5, 8, 0]) = " + allPositive(List.of(2, -5, 8,
0)));
        System.out.println("allPositiveWrong([4, -1])  = " + allPositiveWrong(List.of(4, -
1)));
        check("all non-negative", true, allPositive(List.of(7, 5, 3, 8)));
        check("one negative", false, allPositive(List.of(2, -5, 8, 0)));
        check("zero counts", true, allPositive(List.of(0)));
        check("negative last", false, allPositive(List.of(4, -1)));
        check("empty", true, allPositive(List.nil()));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
allPositive([7, 5, 3, 8])  = true
allPositive([2, -5, 8, 0]) = false
allPositiveWrong([4, -1])  = true
PASS all non-negative
PASS one negative
PASS zero counts
PASS negative last
PASS empty
All 5 checks passed.
```

Listing 8.28 `all_positive.py`: the same check in Python.

```
# all_positive.py - Problem 8.10: are all elements positive (here: >= 0)?
import sys
import unittest
from cons_list import List


def all_positive(a):
    """True when every element is >= 0; True for the empty list."""
    if a.empty():
        return True                         # no element can break the rule
    return a.head() >= 0 and all_positive(a.tail())


def all_positive_wrong(a):
    """Tempting but wrong: stops one element too early."""
    if a.empty():
        return True
    elif a.tail().empty():
        return True                         # the last head is never checked!
    elif a.head() >= 0:
        return all_positive_wrong(a.tail())
    return False


class TestAllPositive(unittest.TestCase):
    def test_cases(self):
        self.assertTrue(all_positive(List.of(7, 5, 3, 8)))
        self.assertFalse(all_positive(List.of(2, -5, 8, 0)))
        self.assertTrue(all_positive(List.of(0)))
        self.assertFalse(all_positive(List.of(4, -1)))
        self.assertTrue(all_positive(List.nil()))


if __name__ == "__main__":
    print("all_positive([7, 5, 3, 8])  =", all_positive(List.of(7, 5, 3, 8)))
    print("all_positive([2, -5, 8, 0]) =", all_positive(List.of(2, -5, 8, 0)))
    print("all_positive_wrong([4, -1]) =", all_positive_wrong(List.of(4, -1)))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
all_positive([7, 5, 3, 8])  = True
all_positive([2, -5, 8, 0]) = False
all_positive_wrong([4, -1]) = True
test_cases (__main__.TestAllPositive.test_cases) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Line by line. `a.head() >= 0 && allPositive(a.tail())` uses *short-circuit evaluation*: if the head is negative, `&&` (Python `and`) knows the answer is false and never makes the recursive call, so the method stops at the first negative element.

A tempting wrong solution. `allPositiveWrong` adds a second base case, "a one-element list is fine", and returns `true` for it *without looking at the element*. Every list whose only negative element is the last one is misjudged: `allPositiveWrong([4, -1])` prints `true`. The extra base case was meant as a shortcut, and it silently skipped a check. The test `negative last` exists to catch exactly this. As a rule, a recursive list method needs one base case, the empty list; a second base case is a warning sign.

> **CHECK YOURSELF.** Write `static boolean anyNegative(List a)`. What must it return for the empty list, and how is it related to `allPositive`?

> *Answer.* `false` for the empty list, and `a.head() < 0 || anyNegative(a.tail())` otherwise. For every list, `anyNegative(a) == !allPositive(a)`.

8.5.4 Problem 8.11 — Keeping only the positive elements

> **PROBLEM.** Include a recursive method `static List positives(List a)` that returns a new list containing all the positive elements of `a` (again, ≥ 0), in the same relative order as in `a`. Example: `[2, 3, -5, 8, -2]` gives `[2, 3, 8]`.

In our own words. This is a *filter*: every element is either kept or dropped, and the kept ones stay in order.

Design. Filter the tail first; then put the head in front of the result if it passes the test, or leave the result as it is if it does not (**Table 8.12**).

Table 8.12 Design of `positives`: a filter keeps order because each kept head is placed in front of the filtered tail.

Part	Answer for `positives`
Empty list	`nil()`
Combine, head ≥ 0	`cons(head, positives(tail))`
Combine, head < 0	`positives(tail)`

Listing 8.29 (Java) and **Listing 8.30** (Python) follow the table; the tests are built into each program.

Listing 8.29 `Positives.java`: filtering the elements that are at least zero.

```java
// Positives.java - Problem 8.11: keep the elements >= 0, in their original order.
public class Positives {

    static List positives(List a) {
        if (a.empty()) {
            return List.nil();
        }
        List rest = positives(a.tail());      // filter the tail first
        return (a.head() >= 0) ? List.cons(a.head(), rest) : rest;
    }

    /** Tempting but wrong: gives up at the first negative element. */
    static List positivesWrong(List a) {
        if (a.empty() || a.head() < 0) {
            return List.nil();
        }
        return List.cons(a.head(), positivesWrong(a.tail()));
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        List a = List.of(2, 3, -5, 8, -2);
        System.out.println("positives(" + a + ")      = " + positives(a));
        System.out.println("positivesWrong(" + a + ") = " + positivesWrong(a));
        check("sample", "[2, 3, 8]", positives(a).toString());
        check("zero kept", "[0, 4]", positives(List.of(-1, 0, 4)).toString());
        check("none", "[]", positives(List.of(-3, -4)).toString());
        check("empty", "[]", positives(List.nil()).toString());
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
positives([2, 3, -5, 8, -2])      = [2, 3, 8]
positivesWrong([2, 3, -5, 8, -2]) = [2, 3]
PASS sample
PASS zero kept
PASS none
PASS empty
All 4 checks passed.
```

Listing 8.30 `positives.py`: the same filter in Python.

```
# positives.py - Problem 8.11: keep the elements >= 0, in their original order.
import sys
import unittest
from cons_list import List


def positives(a):
    if a.empty():
        return List.nil()
    rest = positives(a.tail())            # filter the tail first
    return List.cons(a.head(), rest) if a.head() >= 0 else rest


def positives_wrong(a):
    """Tempting but wrong: gives up at the first negative element."""
    if a.empty() or a.head() < 0:
        return List.nil()
    return List.cons(a.head(), positives_wrong(a.tail()))


class TestPositives(unittest.TestCase):
    def test_cases(self):
        self.assertEqual("[2, 3, 8]", str(positives(List.of(2, 3, -5, 8, -2))))
        self.assertEqual("[0, 4]", str(positives(List.of(-1, 0, 4))))
        self.assertEqual("[]", str(positives(List.of(-3, -4))))
        self.assertEqual("[]", str(positives(List.nil())))


if __name__ == "__main__":
    a = List.of(2, 3, -5, 8, -2)
    print(f"positives({a})       = {positives(a)}")
    print(f"positives_wrong({a}) = {positives_wrong(a)}")
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
positives([2, 3, -5, 8, -2])       = [2, 3, 8]
positives_wrong([2, 3, -5, 8, -2]) = [2, 3]
test_cases (__main__.TestPositives.test_cases) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Line by line. The local variable `rest` holds the filtered tail; the conditional expression then either conses the head onto it or returns it unchanged. Order is preserved without any effort: the head of `a` comes before every element of its tail, and `cons` puts it before every element of the filtered tail.

A tempting wrong solution. `positivesWrong` treats a negative element as the end of the list: `if (a.empty() || a.head() < 0) return nil()`. It returns `[2, 3]` and loses the 8 that followed the -5. This is the list version of using `break` where `continue` was meant. The test with a negative element *in the middle* catches it.

> **CHECK YOURSELF.** Write `static List evens(List a)` that keeps the even elements. What does it return for `[1, 2, 3, 4, -6]`?
>
> *Answer.* Replace the test by `a.head() % 2 == 0`: `[2, 4, -6]`. (In both Java and Python, `-6 % 2` is 0.)

8.5.5 Problem 8.12 — Is the list sorted?

> **PROBLEM.** Include a recursive method `static boolean sorted(List a)` that returns whether `a` is sorted in increasing order. There can be duplicate copies of elements, but sortedness requires that all duplicate copies appear together.

In our own words. "Increasing with duplicates allowed" means *non-decreasing*: every element is less than or equal to the next. If that holds for every neighboring pair, equal elements are automatically together, because anything between two copies of v would have to be both $\geq v$ and $\leq v$, that is, equal to v.

Design. Here the combine step needs *two* elements, the head and the head of the tail, so there are two base cases: the empty list and the one-element list, both sorted (**Table 8.13**). This is the one situation where a second base case is right, because the combine step genuinely looks at two elements.

Table 8.13 Design of `sorted`: each call compares one neighboring pair and delegates the rest.

Part	Answer for `sorted`
0 or 1 element	`true`
Combine	`head <= head(tail) && sorted(tail)`
Edge cases to test	duplicates together `[2,5,5,8]`; duplicates apart `[5,7,5]`

Listing 8.31 (Java) and **Listing 8.32** (Python) follow the table; the tests are built into each program.

Listing 8.31 `SortedCheck.java`: checking non-decreasing order pair by pair.

```java
// SortedCheck.java - Problem 8.12: is the list in increasing (non-decreasing) order?
public class SortedCheck {

    static boolean sorted(List a) {
        if (a.empty() || a.tail().empty()) {
            return true;                        // 0 or 1 element: nothing to compare
        }
        return a.head() <= a.tail().head() && sorted(a.tail());
    }

    /** Tempting but wrong: strict < rejects equal neighbors. */
    static boolean sortedWrong(List a) {
        if (a.empty() || a.tail().empty()) {
            return true;
        }
        return a.head() < a.tail().head() && sortedWrong(a.tail());
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        List dup = List.of(2, 5, 5, 8);
        System.out.println("sorted([2, 3, -5, 8, -2]) = " + sorted(List.of(2, 3, -5, 8, -
2)));
        System.out.println("sorted([2, 3, 8])         = " + sorted(List.of(2, 3, 8)));
        System.out.println("sorted(" + dup + ")      = " + sorted(dup));
        System.out.println("sortedWrong(" + dup + ") = " + sortedWrong(dup));
        check("unsorted", false, sorted(List.of(2, 3, -5, 8, -2)));
        check("sorted", true, sorted(List.of(2, 3, 8)));
        check("duplicates together", true, sorted(dup));
        check("duplicates apart", false, sorted(List.of(5, 7, 5)));
        check("empty", true, sorted(List.nil()));
        check("single", true, sorted(List.of(42)));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
sorted([2, 3, -5, 8, -2]) = false
sorted([2, 3, 8])         = true
sorted([2, 5, 5, 8])      = true
sortedWrong([2, 5, 5, 8]) = false
PASS unsorted
PASS sorted
PASS duplicates together
PASS duplicates apart
PASS empty
PASS single
All 6 checks passed.
```

Listing 8.32 `sorted_check.py`: the same check in Python, named `is_sorted` so as not to hide the built-in `sorted`.

```
# sorted_check.py - Problem 8.12: is the list in increasing (non-decreasing) order?
import sys
import unittest
from cons_list import List


def is_sorted(a):
    if a.empty() or a.tail().empty():
        return True                          # 0 or 1 element: nothing to compare
    return a.head() <= a.tail().head() and is_sorted(a.tail())


def is_sorted_wrong(a):
    """Tempting but wrong: strict < rejects equal neighbors."""
    if a.empty() or a.tail().empty():
        return True
    return a.head() < a.tail().head() and is_sorted_wrong(a.tail())


class TestSorted(unittest.TestCase):
    def test_cases(self):
        self.assertFalse(is_sorted(List.of(2, 3, -5, 8, -2)))
        self.assertTrue(is_sorted(List.of(2, 3, 8)))
        self.assertTrue(is_sorted(List.of(2, 5, 5, 8)))
        self.assertFalse(is_sorted(List.of(5, 7, 5)))
        self.assertTrue(is_sorted(List.nil()))
        self.assertTrue(is_sorted(List.of(42)))


if __name__ == "__main__":
    dup = List.of(2, 5, 5, 8)
    print("is_sorted([2, 3, -5, 8, -2]) =", is_sorted(List.of(2, 3, -5, 8, -2)))
    print("is_sorted([2, 3, 8])         =", is_sorted(List.of(2, 3, 8)))
    print(f"is_sorted({dup})      = {is_sorted(dup)}")
    print(f"is_sorted_wrong({dup}) = {is_sorted_wrong(dup)}")
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
is_sorted([2, 3, -5, 8, -2]) = False
is_sorted([2, 3, 8])         = True
is_sorted([2, 5, 5, 8])      = True
is_sorted_wrong([2, 5, 5, 8]) = False
test_cases (__main__.TestSorted.test_cases) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Line by line. The first `if` covers both base cases in one condition; the `||` guarantees that `a.tail()` is evaluated only when `a` is not empty, so it can never throw `ListException`. The recursive line compares the first pair and asks the tail about the remaining pairs. A list of n elements has $n - 1$ neighboring pairs, and each is compared exactly once.

A tempting wrong solution. `sortedWrong` uses `<` instead of `<=`. It calls `[2, 5, 5, 8]` unsorted, contradicting the statement, which explicitly allows duplicates. Choosing between `<` and `<=` is the most common single-character bug in programming; a test with equal neighbors settles it.

> **CHECK YOURSELF.** Which of these does `sorted` accept: `[]`, `[3, 3, 3]`, `[1, 2, 2, 1]`, `[-4, 0, 7]`?
>
> *Answer.* `[]` true, `[3, 3, 3]` true, `[1, 2, 2, 1]` false (the last pair 2, 1 fails), `[-4, 0, 7]` true.

8.5.6 Problem 8.13 — Merging two sorted lists

> **PROBLEM.** Include a recursive method `static List merge(List a, List b)` that, given two sorted lists, returns a new sorted list containing all the elements of `a` and all the elements of `b`. All duplicates, within `a`, within `b`, or across them, are kept. Examples: `[2, 5, 5, 8]` and `[5, 7, 8, 9]` give `[2, 5, 5, 5, 7, 8, 8, 9]`; `[2, 5, 5, 8]` and `[9]` give `[2, 5, 5, 8, 9]`.

In our own words. The smallest element of the result is the smaller of the two heads. Take it, and merge what remains. When one list runs out, the rest of the other list is the rest of the answer. This is the heart of Merge Sort [5].

Design. Now there are two lists, so there are two base cases, one for each list running out (**Table 8.14**).

Table 8.14 Design of `merge`: each call moves the smaller head to the result, so each call removes one element from one of the lists.

Situation	Answer
`a` empty	`b` (all of it)
`b` empty	`a` (all of it)
`head(a) <= head(b)`	`cons(head(a), merge(tail(a), b))`
otherwise	`cons(head(b), merge(a, tail(b)))`

Each call removes exactly one element from `a` or from `b`, so the number of calls that do work is

$$\text{calls}(\text{merge}(a, b)) \leq |a| + |b|, \tag{8.5}$$

where $|a|$ is the length of a. Equation (8.5) is why merging is fast: its cost is linear in the total length.

Listing 8.33 (Java) and **Listing 8.34** (Python) implement the four cases; the tests are built into each program.

Listing 8.33 `MergeLists.java`: merging two sorted lists with both base cases.

```
// MergeLists.java - Problem 8.13: merge two sorted lists, keeping duplicates.
public class MergeLists {

    static List merge(List a, List b) {
        if (a.empty()) {
            return b;                          // nothing left in a: the rest is b
        }
        if (b.empty()) {
            return a;                          // nothing left in b: the rest is a
        }
        if (a.head() <= b.head()) {
            return List.cons(a.head(), merge(a.tail(), b));
        }
        return List.cons(b.head(), merge(a, b.tail()));
    }

    /** Tempting but wrong: an empty a is taken to mean "nothing left at all". */
    static List mergeWrong(List a, List b) {
        if (a.empty()) {
            return List.nil();
        }
        if (b.empty()) {
            return a;
        }
        if (a.head() <= b.head()) {
            return List.cons(a.head(), mergeWrong(a.tail(), b));
        }
        return List.cons(b.head(), mergeWrong(a, b.tail()));
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }
```

Listing 8.33 (Continued) *MergeLists.java: merging two sorted lists with both base cases.*

```
    public static void main(String[] args) {
        List a = List.of(2, 5, 5, 8);
        List b = List.of(5, 7, 8, 9);
        List c = List.of(9);
        System.out.println("merge(" + a + ", " + b + ") = " + merge(a, b));
        System.out.println("merge(" + a + ", " + c + ") = " + merge(a, c));
        System.out.println("mergeWrong(" + a + ", " + c + ") = " + mergeWrong(a, c));
        check("sample 1", "[2, 5, 5, 5, 7, 8, 8, 9]", merge(a, b).toString());
        check("sample 2", "[2, 5, 5, 8, 9]", merge(a, c).toString());
        check("empty left", "[9]", merge(List.nil(), c).toString());
        check("empty right", a.toString(), merge(a, List.nil()).toString());
        check("both empty", "[]", merge(List.nil(), List.nil()).toString());
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
merge([2, 5, 5, 8], [5, 7, 8, 9]) = [2, 5, 5, 5, 7, 8, 8, 9]
merge([2, 5, 5, 8], [9]) = [2, 5, 5, 8, 9]
mergeWrong([2, 5, 5, 8], [9]) = [2, 5, 5, 8]
PASS sample 1
PASS sample 2
PASS empty left
PASS empty right
PASS both empty
All 5 checks passed.
```

Listing 8.34 `merge_lists.py`: the same merge in Python.

```
# merge_lists.py - Problem 8.13: merge two sorted lists, keeping duplicates.
import sys
import unittest
from cons_list import List


def merge(a, b):
    if a.empty():
        return b                          # nothing left in a: the rest is b
    if b.empty():
        return a                          # nothing left in b: the rest is a
    if a.head() <= b.head():
        return List.cons(a.head(), merge(a.tail(), b))
    return List.cons(b.head(), merge(a, b.tail()))


def merge_wrong(a, b):
    """Tempting but wrong: an empty a is taken to mean "nothing left at all"."""
    if a.empty():
        return List.nil()
    if b.empty():
        return a
    if a.head() <= b.head():
        return List.cons(a.head(), merge_wrong(a.tail(), b))
    return List.cons(b.head(), merge_wrong(a, b.tail()))


class TestMerge(unittest.TestCase):
    def test_samples(self):
        a = List.of(2, 5, 5, 8)
        self.assertEqual("[2, 5, 5, 5, 7, 8, 8, 9]", str(merge(a, List.of(5, 7, 8, 9))))
        self.assertEqual("[2, 5, 5, 8, 9]", str(merge(a, List.of(9))))

    def test_empty_sides(self):
        self.assertEqual("[9]", str(merge(List.nil(), List.of(9))))
        self.assertEqual("[1, 2]", str(merge(List.of(1, 2), List.nil())))
        self.assertEqual("[]", str(merge(List.nil(), List.nil())))


if __name__ == "__main__":
    a, b, c = List.of(2, 5, 5, 8), List.of(5, 7, 8, 9), List.of(9)
    print(f"merge({a}, {b}) = {merge(a, b)}")
    print(f"merge({a}, {c}) = {merge(a, c)}")
    print(f"merge_wrong({a}, {c}) = {merge_wrong(a, c)}")
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
merge([2, 5, 5, 8], [5, 7, 8, 9]) = [2, 5, 5, 5, 7, 8, 8, 9]
merge([2, 5, 5, 8], [9]) = [2, 5, 5, 8, 9]
merge_wrong([2, 5, 5, 8], [9]) = [2, 5, 5, 8]
test_empty_sides (__main__.TestMerge.test_empty_sides) ... ok
test_samples (__main__.TestMerge.test_samples) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Line by line. Trace the second example: heads 2 and 9, take 2; heads 5 and 9, take 5; again 5; then 8; now `a` is empty, so the answer for the rest is `b` itself, `[9]`, returned whole and *shared*, not copied. Sharing is safe because lists are immutable (Section 8.3.2). The `<=` in the third case takes the element from `a` first when the heads are equal; either choice gives the same list of numbers, but `<=` keeps equal elements of `a` ahead of those of `b`, a property called *stability* that matters when elements carry more data than a number.

A tempting wrong solution. `mergeWrong` returns `nil()` when `a` is empty, as if one list running out meant the whole job was done. The output `mergeWrong([2, 5, 5, 8], [9]) = [2, 5, 5, 8]` shows the 9 lost. The four checks named `empty left`, `empty right`, `both empty` and `sample 2` exist because a two-list recursion has four "shapes of emptiness", and each needs a test.

> **CHECK YOURSELF.** How many calls does `merge([1, 3], [2, 4, 6, 8])` make before reaching a base case, and what is the result?
>
> *Answer.* The heads taken are 1, 2, 3, 4; after the fourth call `a` is empty, and `[6, 8]` is returned whole. Four working calls (fewer than $|a| + |b| = 6$), result `[1, 2, 3, 4, 6, 8]`.

8.5.7 Problem 8.14 — Skipping every copy of an element

> **PROBLEM.** Include a recursive method `static List skip(int x, List a)` that returns a copy of `a` with every occurrence of `x` skipped. Example: `x = 5`, `a = [2, 5, 5, 5, 7, 8, 8, 9]` gives `[2, 7, 8, 8, 9]`. (This method is needed in Problem 8.15.)

In our own words. `skip` is a filter, like `positives`, whose test is "the element is not `x`". We solve it before removing duplicates because Problem 8.15 uses it.

Design. **Table 8.15** is **Table 8.12** with a different test.

Table 8.15 Design of `skip`: a filter whose test is "not equal to `x`".

Part	Answer for `skip`
Empty list	`nil()`
Combine, head `== x`	`skip(x, tail)`
Combine, head `!= x`	`cons(head, skip(x, tail))`
Edge cases to test	`x` last; `x` absent; list made only of `x`

Listing 8.35 (Java) and **Listing 8.36** (Python) follow the table; the tests are built into each program.

Listing 8.35 `SkipElement.java`: removing every copy of one value.

```java
// SkipElement.java - Problem 8.14: a copy of the list without any copy of x.
public class SkipElement {

    static List skip(int x, List a) {
        if (a.empty()) {
            return List.nil();
        }
        List rest = skip(x, a.tail());
        return (a.head() == x) ? rest : List.cons(a.head(), rest);
    }

    /** Tempting but wrong: a one-element list is copied without looking at it. */
    static List skipWrong(int x, List a) {
        if (a.empty()) {
            return List.nil();
        }
        if (a.tail().empty()) {
            return List.cons(a.head(), List.nil());
        }
        List rest = skipWrong(x, a.tail());
        return (a.head() == x) ? rest : List.cons(a.head(), rest);
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        List a = List.of(2, 5, 5, 5, 7, 8, 8, 9);
        System.out.println("skip(5, a)      = " + skip(5, a));
        System.out.println("skip(9, a)      = " + skip(9, a));
        System.out.println("skipWrong(9, a) = " + skipWrong(9, a));
        check("sample", "[2, 7, 8, 8, 9]", skip(5, a).toString());
        check("last element", "[2, 5, 5, 5, 7, 8, 8]", skip(9, a).toString());
        check("absent", a.toString(), skip(4, a).toString());
        check("only x", "[]", skip(3, List.of(3, 3)).toString());
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
skip(5, a)      = [2, 7, 8, 8, 9]
skip(9, a)      = [2, 5, 5, 5, 7, 8, 8]
skipWrong(9, a) = [2, 5, 5, 5, 7, 8, 8, 9]
PASS sample
PASS last element
PASS absent
PASS only x
All 4 checks passed.
```

Listing 8.36 `skip_element.py`: the same filter in Python.

```python
# skip_element.py - Problem 8.14: a copy of the list without any copy of x.
import sys
import unittest
from cons_list import List


def skip(x, a):
    if a.empty():
        return List.nil()
    rest = skip(x, a.tail())
    return rest if a.head() == x else List.cons(a.head(), rest)


def skip_wrong(x, a):
    """Tempting but wrong: a one-element list is copied without looking at it."""
    if a.empty():
        return List.nil()
    if a.tail().empty():
        return List.cons(a.head(), List.nil())
    rest = skip_wrong(x, a.tail())
    return rest if a.head() == x else List.cons(a.head(), rest)


class TestSkip(unittest.TestCase):
    def test_cases(self):
        a = List.of(2, 5, 5, 5, 7, 8, 8, 9)
        self.assertEqual("[2, 7, 8, 8, 9]", str(skip(5, a)))
        self.assertEqual("[2, 5, 5, 5, 7, 8, 8]", str(skip(9, a)))
        self.assertEqual(str(a), str(skip(4, a)))
        self.assertEqual("[]", str(skip(3, List.of(3, 3))))


if __name__ == "__main__":
    a = List.of(2, 5, 5, 5, 7, 8, 8, 9)
    print("skip(5, a)       =", skip(5, a))
    print("skip(9, a)       =", skip(9, a))
    print("skip_wrong(9, a) =", skip_wrong(9, a))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
skip(5, a)       = [2, 7, 8, 8, 9]
skip(9, a)       = [2, 5, 5, 5, 7, 8, 8]
skip_wrong(9, a) = [2, 5, 5, 5, 7, 8, 8, 9]
test_cases (__main__.TestSkip.test_cases) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Line by line. The method is two lines of real work: filter the tail, then keep or drop the head. The check `only x` shows that a list made entirely of `x` becomes empty.

A tempting wrong solution. `skipWrong` repeats the pattern we saw in Problem 8.10: an extra base case copies a one-element list *without testing its element*. Every `x` in the last position survives: `skipWrong(9, a)` still ends with 9. Notice that `skip(5, a)`, the example of the statement, gives the right answer for both versions; only an edge-case test (`last element`) reveals the bug.

> **CHECK YOURSELF.** What does `skip(8, [8, 2, 8])` return, and how many recursive calls are made in total (including the one on the empty list)?
>
> *Answer.* `[2]`. Four calls: on `[8, 2, 8]`, `[2, 8]`, `[8]` and `[]`, one per element plus one for the empty list.

8.5.8 Problem 8.15 — Removing duplicates

> **PROBLEM.** Include a recursive method `static List removeDuplicates(List a)` that returns a copy of `a` with all duplicate copies removed. Example: `[2, 5, 5, 5, 7, 8, 8, 9]` gives `[2, 5, 7, 8, 9]`. Hint: define helper methods, such as `skip` of Problem 8.14.

In our own words. Keep the first copy of every value and delete the later ones. The statement's example happens to be sorted, but nothing in the statement says the input must be, so the method must also work on `[3, 1, 3, 2, 1]`, giving `[3, 1, 2]`.

Design. Keep the head; remove every later copy of it from the tail with `skip`; then remove duplicates from what is left (**Table 8.16**).

Table 8.16 Design of `removeDuplicates`: the head survives, its later copies are skipped, and the recursion handles the rest.

Part	Answer for `removeDuplicates`
Empty list	`nil()` (no elements, no duplicates)
Combine	`cons(head, removeDuplicates(skip(head, tail)))`
Helper	`SkipElement.skip` from Problem 8.14, reused, not copied

Listing 8.37 (Java) and **Listing 8.38** (Python) follow the table; the tests are built into each program.

Listing 8.37 `RemoveDuplicates.java`: removing duplicates with the help of `skip`.

```java
// RemoveDuplicates.java - Problem 8.15: keep only the first copy of every element.
public class RemoveDuplicates {

    /** Keep the head, delete its later copies (skip), then repeat on the rest. */
    static List removeDuplicates(List a) {
        if (a.empty()) {
            return List.nil();                // no elements, no duplicates
        }
        List restWithoutHead = SkipElement.skip(a.head(), a.tail());
        return List.cons(a.head(), removeDuplicates(restWithoutHead));
    }

    /** Tempting but wrong: only compares neighbors, so it needs a sorted list. */
    static List removeDuplicatesWrong(List a) {
        if (a.empty() || a.tail().empty()) {
            return a;
        }
        if (a.head() == a.tail().head()) {
            return removeDuplicatesWrong(a.tail());
        }
        return List.cons(a.head(), removeDuplicatesWrong(a.tail()));
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        List sortedList = List.of(2, 5, 5, 5, 7, 8, 8, 9);
        List unsorted = List.of(3, 1, 3, 2, 1);
        System.out.println("removeDuplicates(" + sortedList + ") = " +
removeDuplicates(sortedList));
        System.out.println("removeDuplicates(" + unsorted + ")          = " +
removeDuplicates(unsorted));
        System.out.println("removeDuplicatesWrong(" + unsorted + ")     = " +
removeDuplicatesWrong(unsorted));
        check("sorted sample", "[2, 5, 7, 8, 9]", removeDuplicates(sortedList).toString());
        check("unsorted", "[3, 1, 2]", removeDuplicates(unsorted).toString());
        check("empty", "[]", removeDuplicates(List.nil()).toString());
        check("all equal", "[4]", removeDuplicates(List.of(4, 4, 4)).toString());
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
removeDuplicates([2, 5, 5, 5, 7, 8, 8, 9]) = [2, 5, 7, 8, 9]
removeDuplicates([3, 1, 3, 2, 1])          = [3, 1, 2]
removeDuplicatesWrong([3, 1, 3, 2, 1])     = [3, 1, 3, 2, 1]
PASS sorted sample
PASS unsorted
PASS empty
PASS all equal
All 4 checks passed.
```

Listing 8.38 `remove_duplicates.py`: the same method in Python, importing `skip` from Problem 8.14.

```python
# remove_duplicates.py - Problem 8.15: keep only the first copy of every element.
import sys
import unittest
from cons_list import List
from skip_element import skip


def remove_duplicates(a):
    """Keep the head, delete its later copies (skip), then repeat on the rest."""
    if a.empty():
        return List.nil()                # no elements, no duplicates
    rest_without_head = skip(a.head(), a.tail())
    return List.cons(a.head(), remove_duplicates(rest_without_head))


def remove_duplicates_wrong(a):
    """Tempting but wrong: only compares neighbors, so it needs a sorted list."""
    if a.empty() or a.tail().empty():
        return a
    if a.head() == a.tail().head():
        return remove_duplicates_wrong(a.tail())
    return List.cons(a.head(), remove_duplicates_wrong(a.tail()))


class TestRemoveDuplicates(unittest.TestCase):
    def test_cases(self):
        self.assertEqual("[2, 5, 7, 8, 9]",
                         str(remove_duplicates(List.of(2, 5, 5, 5, 7, 8, 8, 9))))
        self.assertEqual("[3, 1, 2]", str(remove_duplicates(List.of(3, 1, 3, 2, 1))))
        self.assertEqual("[]", str(remove_duplicates(List.nil())))
        self.assertEqual("[4]", str(remove_duplicates(List.of(4, 4, 4))))


if __name__ == "__main__":
    s, u = List.of(2, 5, 5, 5, 7, 8, 8, 9), List.of(3, 1, 3, 2, 1)
    print(f"remove_duplicates({s}) = {remove_duplicates(s)}")
    print(f"remove_duplicates({u})          = {remove_duplicates(u)}")
    print(f"remove_duplicates_wrong({u})    = {remove_duplicates_wrong(u)}")
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
remove_duplicates([2, 5, 5, 5, 7, 8, 8, 9]) = [2, 5, 7, 8, 9]
remove_duplicates([3, 1, 3, 2, 1])          = [3, 1, 2]
remove_duplicates_wrong([3, 1, 3, 2, 1])    = [3, 1, 3, 2, 1]
test_cases (__main__.TestRemoveDuplicates.test_cases) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Line by line. For `[3, 1, 3, 2, 1]`: keep 3, skip the other 3 from `[1, 3, 2, 1]` to get `[1, 2, 1]`; keep 1, skip 1 from `[2, 1]` to get `[2]`; keep 2. The result is `[3, 1, 2]`. The Java program calls `SkipElement.skip(...)`, a static method of another class in the same folder, and the Python program imports it with `from skip_element import skip`: the helper is *reused*, so a bug fixed in one place is fixed everywhere.

The price of this simplicity is time. On a list of n distinct elements, the first call to `skip` walks $n-1$ elements, the next $n-2$, and so on, so the total work is

$$(n-1)+(n-2)+\cdots+1+0=\frac{n(n-1)}{2}, \tag{8.6}$$

which grows like n^2. For the lists of this chapter that is nothing; for a million customer records, Equation (8.6) gives about 5×10^{11} steps, and a hash set (Chapter 9) would be the right tool.

A tempting wrong solution. `removeDuplicatesWrong` compares each element only with its neighbor, which is correct for sorted input and is what the example suggests. On the unsorted list it removes nothing: `[3, 1, 3, 2, 1]`. A second trap is to throw an exception for the empty list, on the feeling that "there is nothing to do". An empty list has no duplicates, and the correct answer is the empty list; exceptions are for situations that have no sensible answer, as `head()` of the empty list has none.

> **CHECK YOURSELF.** What does `removeDuplicates([4, 4, 1, 4, 1, 7])` return? Using Equation (8.6) as a guide, is the number of elements walked by the `skip` calls more or less than $\frac{6 \cdot 5}{2} = 15$?
>
> *Answer.* `[4, 1, 7]`. Less: `skip(4, [4, 1, 4, 1, 7])` walks 5 elements and leaves `[1, 1, 7]`; then `skip(1, [1, 7])` walks 2 and leaves `[7]`; then `skip(7, [])` walks 0. The total is 7, well below 15, because every removed copy shortens all the later walks.

8.6 New Worked Problems: Exceptions, Files, and Arrays

The remaining six problems are new, and each is set in the running case study of this book, Profitina Laundry. They are larger than the list problems because each involves several classes, so each one begins with the method of Section 8.2.

8.6.1 Problem 8.16 — An exception-safe order file parser

> **PROBLEM.** Profitina Laundry keeps its day's orders in a text file, one order per line in the form `id;customer;service;kg`, where the service is `WASH`, `IRON` or `EXPRESS` and the weight is a positive number. Lines starting with `#` and empty lines are ignored. Write a program that reads the file, keeps every valid order, and reports every invalid line with its line number and the reason, *without stopping at the first bad line*. A missing file must produce a clear message, not a crash.

In our own words. Two different things can go wrong, and they deserve different treatment. A *bad line* is expected: files typed by people contain mistakes, and the right reaction is to note the mistake and carry on. A *missing file* is a failure of the whole job; there is nothing to carry on with, so the caller must decide what to do.

Design. Steps 1 to 5 of the method give three classes and two methods (**Table 8.17**). The sample file, `laundry_orders.txt`, sits next to the program and contains one valid order and one mistake of every kind:

```
# id;customer;service;kg
A101;Sari;WASH;3.5
A102;Budi;IRON;2

A103;Dewi;EXPRESS;abc
A104;;WASH;4
A105;Rudi;DRYCLEAN;1.5
A106;Tono;WASH;-2
A107;Ayu;EXPRESS;5
A108;Lina;WASH
```

Table 8.17 Design of the order parser: each kind of failure is reported by the method that detects it and handled by the method that can do something about it.

Class or method	Responsibility	Failure it reports
`Order`	holds one valid order; immutable	none: it is only built from valid data
`OrderFormatException`	says which line is bad and why	checked; extends `Exception`
`parseLine(line, lineNo)`	one line to one `Order`	throws `OrderFormatException`
`readOrders(file, errors)`	the whole file, line by line	collects bad lines; throws `IOException` for the file
`main`	prints the report	catches `NoSuchFileException`

The decisive design choice is *where* each `try` goes. The `try` that catches `OrderFormatException` is *inside* the loop, around one line, so one bad line costs one line. The file itself is opened with `try`-with-resources (Java) or `with` (Python), so it is closed whatever happens [6], [7].

The Java solution. **Listing 8.39** is the longest program of the chapter, and every part of it is a Chapter 5 idea.

Listing 8.39 `LaundryOrderParser.java`: a line parser with a custom checked exception, a file reader with try-with-resources, and a wrong reader for comparison.

```java
// LaundryOrderParser.java - Problem 8.16: an exception-safe parser for an order file.
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.NoSuchFileException;
import java.nio.file.Path;
import java.util.ArrayList;

public class LaundryOrderParser {

    /** A checked exception: callers of parseLine must decide what to do. */
    static final class OrderFormatException extends Exception {
        private static final long serialVersionUID = 1L;

        OrderFormatException(int lineNo, String reason) {
            super("line " + lineNo + ": " + reason);
        }
    }

    /** One valid order. All fields are private and final: an order never changes. */
    static final class Order {
        private final String id;
        private final String customer;
        private final String service;
        private final double kg;

        Order(String id, String customer, String service, double kg) {
            this.id = id;
            this.customer = customer;
            this.service = service;
            this.kg = kg;
        }

        double getKg() {
            return kg;
        }

        @Override
        public String toString() {
            return id + " " + customer + " " + service + " " + kg + " kg";
        }
    }

    private static final String[] SERVICES = {"WASH", "IRON", "EXPRESS"};
```

Listing 8.39 (Continued) *LaundryOrderParser.java: a line parser with a custom checked exception, a file reader with try-with-resources, and a wrong reader for comparison.*

```
    /** Turns one line "id;customer;service;kg" into an Order, or explains why not. */
    static Order parseLine(String line, int lineNo) throws OrderFormatException {
        String[] parts = line.split(";", -1);          // -1 keeps empty fields
        if (parts.length != 4) {
            throw new OrderFormatException(lineNo, "expected 4 fields, found " +
parts.length);
        }
        String id = parts[0].trim();
        String customer = parts[1].trim();
        String service = parts[2].trim();
        if (customer.isEmpty()) {
            throw new OrderFormatException(lineNo, "customer name is missing");
        }
        if (!java.util.Arrays.asList(SERVICES).contains(service)) {
            throw new OrderFormatException(lineNo, "unknown service '" + service + "'");
        }
        double kg;
        try {
            kg = Double.parseDouble(parts[3].trim());
        } catch (NumberFormatException e) {            // translate the low-level error
            throw new OrderFormatException(lineNo, "weight '" + parts[3].trim() + "' is not
a number");
        }
        if (kg <= 0) {
            throw new OrderFormatException(lineNo, "weight must be positive, got " + kg);
        }
        return new Order(id, customer, service, kg);
    }

    /**
     * Reads every order in the file. Bad lines are reported in errors and skipped;
     * a missing or unreadable file is the caller's problem (IOException).
     */
    static ArrayList<Order> readOrders(Path file, ArrayList<String> errors) throws
IOException {
        ArrayList<Order> orders = new ArrayList<>();
        try (BufferedReader in = Files.newBufferedReader(file)) {   // closed automatically
            String line;
            int lineNo = 0;
            while ((line = in.readLine()) != null) {
                lineNo++;
                if (line.isBlank() || line.startsWith("#")) {
                    continue;                                  // comment or empty line
                }
                try {
                    orders.add(parseLine(line, lineNo));
                } catch (OrderFormatException e) {
                    errors.add(e.getMessage());                // record it, keep going
                }
```

Listing 8.39 (Continued) *LaundryOrderParser.java: a line parser with a custom checked exception, a file reader with try-with-resources, and a wrong reader for comparison.*

```
            }
        }
        return orders;
    }

    /** Tempting but wrong: one try around the whole loop, so the first bad line ends the
reading. */
    static ArrayList<Order> readOrdersWrong(Path file) throws IOException {
        ArrayList<Order> orders = new ArrayList<>();
        try (BufferedReader in = Files.newBufferedReader(file)) {
            String line;
            int lineNo = 0;
            while ((line = in.readLine()) != null) {
                lineNo++;
                if (!line.isBlank() && !line.startsWith("#")) {
                    orders.add(parseLine(line, lineNo));
                }
            }
        } catch (OrderFormatException e) {
            System.out.println("wrong reader gave up at " + e.getMessage());
        }
        return orders;
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) throws IOException, OrderFormatException {
        String fileName = (args.length > 0) ? args[0] : "laundry_orders.txt";
        ArrayList<String> errors = new ArrayList<>();
        try {
            ArrayList<Order> orders = readOrders(Path.of(fileName), errors);
            double total = 0;
            for (Order o : orders) {
                System.out.println("OK    " + o);
                total += o.getKg();
            }
            for (String e : errors) {
                System.out.println("ERROR " + e);
            }
```

Listing 8.39 (Continued) *LaundryOrderParser.java: a line parser with a custom checked exception, a file reader with try-with-resources, and a wrong reader for comparison.*

```
            System.out.println(orders.size() + " orders, " + errors.size() + " errors, " +
total + " kg");
            System.out.println("wrong reader kept " +
readOrdersWrong(Path.of(fileName)).size() + " orders");
        } catch (NoSuchFileException e) {
            System.out.println("Cannot find the file " + e.getFile());
        }

        check("valid line", "A101 Sari WASH 3.5 kg", parseLine("A101;Sari;WASH;3.5",
1).toString());
        try {
            parseLine("A103;Dewi;EXPRESS;abc", 5);
            throw new AssertionError("no exception");
        } catch (OrderFormatException e) {
            check("bad number", "line 5: weight 'abc' is not a number", e.getMessage());
        }
        try {
            readOrders(Path.of("no_such_file.txt"), new ArrayList<>());
            throw new AssertionError("no exception");
        } catch (NoSuchFileException e) {
            check("missing file", "no_such_file.txt", e.getFile());
        }
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
OK    A101 Sari WASH 3.5 kg
OK    A102 Budi IRON 2.0 kg
OK    A107 Ayu EXPRESS 5.0 kg
ERROR line 5: weight 'abc' is not a number
ERROR line 6: customer name is missing
ERROR line 7: unknown service 'DRYCLEAN'
ERROR line 8: weight must be positive, got -2.0
ERROR line 10: expected 4 fields, found 3
3 orders, 5 errors, 10.5 kg
wrong reader gave up at line 5: weight 'abc' is not a number
wrong reader kept 2 orders
PASS valid line
PASS bad number
PASS missing file
All 3 checks passed.
```

The Python solution. **Listing 8.40** has the same structure; Python's `with open(...)` plays the role of try-with-resources.

Listing 8.40 `laundry_order_parser.py`: the same parser in Python, with `unittest` tests.

```
# laundry_order_parser.py - Problem 8.16: an exception-safe parser for an order file.
import sys
import unittest

SERVICES = ("WASH", "IRON", "EXPRESS")


class OrderFormatError(Exception):
    """One line of the order file is malformed."""

    def __init__(self, line_no, reason):
        super().__init__(f"line {line_no}: {reason}")


class Order:
    """One valid order. Attributes are 'private by convention' and read-only."""

    def __init__(self, order_id, customer, service, kg):
        self._id = order_id
        self._customer = customer
        self._service = service
        self._kg = kg

    @property
    def kg(self):
        return self._kg

    def __str__(self):
        return f"{self._id} {self._customer} {self._service} {self._kg} kg"


def parse_line(line, line_no):
    """Turns one line "id;customer;service;kg" into an Order, or explains why not."""
    parts = line.split(";")
    if len(parts) != 4:
        raise OrderFormatError(line_no, f"expected 4 fields, found {len(parts)}")
    order_id, customer, service, weight = (p.strip() for p in parts)
    if not customer:
        raise OrderFormatError(line_no, "customer name is missing")
    if service not in SERVICES:
        raise OrderFormatError(line_no, f"unknown service '{service}'")
    try:
        kg = float(weight)
    except ValueError:                          # translate the low-level error
        raise OrderFormatError(line_no, f"weight '{weight}' is not a number") from None
    if kg <= 0:
```

Listing 8.40 (Continued) *laundry_order_parser.py: the same parser in Python, with unittest tests.*

```
        raise OrderFormatError(line_no, f"weight must be positive, got {kg}")
    return Order(order_id, customer, service, kg)


def read_orders(file_name, errors):
    """Reads every order in the file. Bad lines are reported in errors and
    skipped; a missing file raises FileNotFoundError for the caller."""
    orders = []
    with open(file_name, encoding="utf-8") as f:      # closed automatically
        for line_no, line in enumerate(f, start=1):
            line = line.rstrip("\n")
            if not line.strip() or line.startswith("#"):
                continue                              # comment or empty line
            try:
                orders.append(parse_line(line, line_no))
            except OrderFormatError as e:
                errors.append(str(e))                 # record it, keep going
    return orders


class TestParser(unittest.TestCase):
    def test_valid_line(self):
        self.assertEqual("A101 Sari WASH 3.5 kg", str(parse_line("A101;Sari;WASH;3.5", 1)))

    def test_bad_number(self):
        with self.assertRaises(OrderFormatError) as ctx:
            parse_line("A103;Dewi;EXPRESS;abc", 5)
        self.assertEqual("line 5: weight 'abc' is not a number", str(ctx.exception))

    def test_missing_file(self):
        with self.assertRaises(FileNotFoundError):
            read_orders("no_such_file.txt", [])


def main(file_name):
    errors = []
    try:
        orders = read_orders(file_name, errors)
    except FileNotFoundError:
        print("Cannot find the file", file_name)
        return
    for o in orders:
        print("OK   ", o)
    for e in errors:
        print("ERROR", e)
    print(f"{len(orders)} orders, {len(errors)} errors, {sum(o.kg for o in orders)} kg")
```

Listing 8.40 (Continued) *laundry_order_parser.py: the same parser in Python, with unittest tests.*

```


if __name__ == "__main__":
    main(sys.argv[1] if len(sys.argv) > 1 else "laundry_orders.txt")
    sys.stdout.flush()
    unittest.main(argv=sys.argv[:1], verbosity=2)
```

```
OK    A101 Sari WASH 3.5 kg
OK    A102 Budi IRON 2.0 kg
OK    A107 Ayu EXPRESS 5.0 kg
ERROR line 5: weight 'abc' is not a number
ERROR line 6: customer name is missing
ERROR line 7: unknown service 'DRYCLEAN'
ERROR line 8: weight must be positive, got -2.0
ERROR line 10: expected 4 fields, found 3
3 orders, 5 errors, 10.5 kg
test_bad_number (__main__.TestParser.test_bad_number) ... ok
test_missing_file (__main__.TestParser.test_missing_file) ... ok
test_valid_line (__main__.TestParser.test_valid_line) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Line by line. `OrderFormatException` extends `Exception`, not `RuntimeException`, so it is *checked*: `parseLine` must declare `throws OrderFormatException`, and every caller must either catch it or declare it in turn. That is deliberate. A bad line is a normal event that callers must think about, and the compiler now makes sure they do. `line.split(";", -1)` splits on semicolons; the `-1` keeps empty trailing fields, so a line ending in `;` is counted correctly. In the weight conversion, `Double.parseDouble` throws the library's `NumberFormatException` for `abc`; the `catch` *translates* it into our own exception with the line number and a sentence a person can act on, which is far more useful than `For input string: "abc"`. In `readOrders`, the `try (BufferedReader in = ...)` header opens the file and guarantees `in.close()` when the block ends, normally or by an exception. `Files.newBufferedReader` throws `NoSuchFileException` (a subclass of `IOException`) for a missing file, and `main` turns it into the message `Cannot find the file ...`: run `java LaundryOrderParser nothing.txt` to see it. The report shows three valid orders totalling 10.5 kg and five errors, one for each kind of mistake in the file; the blank line 4 and the comment line 1 are skipped silently. The three checks test a valid line, a bad number with its exact message, and a missing file.

In Python, `float("abc")` raises `ValueError`, which we translate with `raise ... from None`; the `from None` hides the original error from the traceback, because our message already says everything. `open` raises `FileNotFoundError` for a missing file. The `unittest.main(argv=sys.argv[:1], ...)` call passes only the program name to `unittest`, so that a file name given on the command line is not mistaken for a test name.

A tempting wrong solution. `readOrdersWrong` puts one `try` around the *whole* loop. It is shorter, and it compiles. But the first bad line throws an exception out of the loop, the `catch` runs, and the reading is over: the output says `wrong reader gave up at line 5` and `wrong reader kept 2 orders`, losing the valid order `A107` that came later. A second classic mistake is `catch (Exception e) {}`, an empty catch-all that hides every error, including programming errors such as a `NullPointerException`. Catch the most specific exception you can handle, as close as possible to where you can handle it.

> **CHECK YOURSELF.** A new rule says that one order may not exceed 20 kg. Where does the check go, what does the report say for the line `A109;Nia;WASH;25`, and does any other method change?
>
> *Answer.* In `parseLine`, next to the positive-weight test: `if (kg > 20) throw new OrderFormatException(lineNo, "weight " + kg + " exceeds 20 kg")`. The report would

contain `ERROR line 11: weight 25.0 exceeds 20 kg` (Python prints `25.0` as well). Nothing else changes, because only `parseLine` knows the rules for a line.

8.6.2 Problem 8.17 — A documented array that grows itself

PROBLEM. Java arrays have a fixed length. Write a class `GrowableIntArray` that stores `int` values in an ordinary array and doubles the array's capacity whenever it is full. It offers `add(value)`, `get(index)`, `size()` and `capacity()`. `get` must reject any index outside `0` to `size() - 1` with an exception. Document every public method with Javadoc (Python: docstrings) and test the class, including the boundaries.

In our own words. This is how `ArrayList` works inside (Chapter 5), built by hand. The class distinguishes two numbers: the *size*, how many values the user has added, and the *capacity*, how many cells the underlying array has. Only the first `size` cells hold user data; the rest are spare room.

Design. Two private fields, `int[] data` and `int size`, and one rule, called a *class invariant* because every public method must preserve it:

$$0 \le \mathit{size} \le \mathit{capacity} = \texttt{data.length}, \quad \text{and only } \texttt{data}[0..\mathit{size}-1] \text{ holds values.} \tag{8.7}$$

Table 8.18 lists what each method does to keep Equation (8.7) true.

Table 8.18 Design of `GrowableIntArray`. The precondition of `get` is checked against `size`, not against the array length.

Method	Precondition	What it does	Exception
constructor	capacity ≥ 1	`data = new int[capacity]`, `size = 0`	`IllegalArgumentException`
`add(v)`	none	if full, copy into an array twice as long; then store, `size++`	none
`get(i)`	$0 \le i < \mathit{size}$	return `data[i]`	`IndexOutOfBoundsException`
`size()`, `capacity()`	none	report the two numbers	none

The Java solution. In **Listing 8.41** every public member has a Javadoc comment. Running `javadoc -d docs GrowableIntArray.java` turns them into web pages; with the `-quiet` option it prints nothing at all for this file, which means no comment is missing a `@param`, `@return` or `@throws` tag.

Listing 8.41 `GrowableIntArray.java`: a self-doubling array with complete Javadoc and boundary checks.

```
// GrowableIntArray.java - Problem 8.17: a documented, tested array that grows itself.
import java.util.Arrays;

/**
 * A list of {@code int} values stored in an ordinary array that doubles its
 * capacity whenever it is full. Only the first {@link #size()} cells hold data.
 */
public class GrowableIntArray {

    private int[] data;
    private int size;

    /**
     * Creates an empty array with room for {@code initialCapacity} values.
     *
     * @param initialCapacity the starting capacity, at least 1
     * @throws IllegalArgumentException if {@code initialCapacity < 1}
     */
    public GrowableIntArray(int initialCapacity) {
        if (initialCapacity < 1) {
            throw new IllegalArgumentException("capacity must be at least 1: " +
initialCapacity);
        }
        data = new int[initialCapacity];
        size = 0;
    }

    /**
     * Appends {@code value} at the end, doubling the capacity first if necessary.
     *
     * @param value the value to append
     */
    public void add(int value) {
        if (size == data.length) {
            data = Arrays.copyOf(data, 2 * data.length);   // new, bigger array
        }
        data[size] = value;
        size++;
    }
```

Listing 8.41 (Continued) *GrowableIntArray.java: a self-doubling array with complete Javadoc and boundary checks.*

```
    /**
     * Returns the value at position {@code index}.
     *
     * @param index a position from 0 to {@code size() - 1}
     * @return the value stored there
     * @throws IndexOutOfBoundsException if {@code index} is outside that range
     */
    public int get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("index " + index + ", size " + size);
        }
        return data[index];
    }

    /** Tempting but wrong: {@code >} instead of {@code >=} lets index == size through. */
    int getWrong(int index) {
        if (index < 0 || index > size) {
            throw new IndexOutOfBoundsException("index " + index + ", size " + size);
        }
        return data[index];
    }

    /**
     * Tells how many values have been added.
     *
     * @return the number of values stored
     */
    public int size() {
        return size;
    }

    /**
     * Tells how many values fit before the next doubling.
     *
     * @return the length of the underlying array
     */
    public int capacity() {
        return data.length;
    }

    @Override
    public String toString() {
        return Arrays.toString(Arrays.copyOf(data, size));  // only the used part
    }
```

Listing 8.41 (Continued) *GrowableIntArray.java: a self-doubling array with complete Javadoc and boundary checks.*

```
    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    /**
     * Demonstrates the class and runs its checks.
     *
     * @param args not used
     */
    public static void main(String[] args) {
        GrowableIntArray a = new GrowableIntArray(2);
        for (int kg = 3; kg <= 7; kg++) {
            a.add(kg);
            System.out.println("add(" + kg + ") -> " + a + "  size " + a.size() + ",
capacity " + a.capacity());
        }
        check("values", "[3, 4, 5, 6, 7]", a.toString());
        check("get(0)", 3, a.get(0));
        check("get(4)", 7, a.get(4));
        check("capacity doubled twice", 8, a.capacity());
        for (int bad : new int[] {5, -1}) {
            try {
                a.get(bad);
                throw new AssertionError("no exception for " + bad);
            } catch (IndexOutOfBoundsException e) {
                check("get(" + bad + ") rejected", "index " + bad + ", size 5",
e.getMessage());
            }
        }
        try {
            new GrowableIntArray(0);
        } catch (IllegalArgumentException e) {
            check("capacity 0 rejected", "capacity must be at least 1: 0", e.getMessage());
        }
        System.out.println("getWrong(5) = " + a.getWrong(5) + "   <- a value that was never
added");
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
add(3) -> [3]  size 1, capacity 2
add(4) -> [3, 4]  size 2, capacity 2
add(5) -> [3, 4, 5]  size 3, capacity 4
add(6) -> [3, 4, 5, 6]  size 4, capacity 4
add(7) -> [3, 4, 5, 6, 7]  size 5, capacity 8
PASS values
PASS get(0)
PASS get(4)
PASS capacity doubled twice
PASS get(5) rejected
PASS get(-1) rejected
PASS capacity 0 rejected
getWrong(5) = 0   <- a value that was never added
All 7 checks passed.
```

The Python solution. **Listing 8.42** imitates a fixed block with `[0] * capacity`, because a Python list would grow by itself and hide the lesson. The docstrings follow the common "Raises:" layout, and `help(GrowableIntArray.get)` prints them.

Listing 8.42 `growable_int_array.py`: the same class in Python, with docstrings and a `setUp` method shared by the tests.

```
# growable_int_array.py - Problem 8.17: a documented, tested array that grows itself.
import sys
import unittest


class GrowableIntArray:
    """A list of ints stored in a fixed-size block that doubles its capacity
    whenever it is full. Only the first size() cells hold data."""

    def __init__(self, initial_capacity):
        """Create an empty array with room for initial_capacity values.

        Raises:
            ValueError: if initial_capacity < 1.
        """
        if initial_capacity < 1:
            raise ValueError(f"capacity must be at least 1: {initial_capacity}")
        self._data = [0] * initial_capacity     # a fixed block, like new int[n]
        self._size = 0

    def add(self, value):
        """Append value at the end, doubling the capacity first if necessary."""
        if self._size == len(self._data):
            self._data = self._data + [0] * len(self._data)   # new, bigger block
        self._data[self._size] = value
        self._size += 1

    def get(self, index):
        """Return the value at position index (0 to size() - 1).

        Raises:
            IndexError: if index is outside that range.
        """
        if index < 0 or index >= self._size:
            raise IndexError(f"index {index}, size {self._size}")
        return self._data[index]

    def size(self):
        """How many values have been added."""
        return self._size

    def capacity(self):
        """How many values fit before the next doubling."""
        return len(self._data)
```

Listing 8.42 (Continued) *growable_int_array.py: the same class in Python, with docstrings and a setUp method shared by the tests.*

```
    def __str__(self):
        return str(self._data[:self._size])        # only the used part


class TestGrowableIntArray(unittest.TestCase):
    def setUp(self):                                # runs before every test
        self.a = GrowableIntArray(2)
        for kg in range(3, 8):
            self.a.add(kg)

    def test_values_and_capacity(self):
        self.assertEqual("[3, 4, 5, 6, 7]", str(self.a))
        self.assertEqual(7, self.a.get(4))
        self.assertEqual(8, self.a.capacity())

    def test_bad_index(self):
        for bad in (5, -1):
            with self.assertRaises(IndexError):
                self.a.get(bad)

    def test_bad_capacity(self):
        with self.assertRaises(ValueError):
            GrowableIntArray(0)


if __name__ == "__main__":
    a = GrowableIntArray(2)
    for kg in range(3, 8):
        a.add(kg)
        print(f"add({kg}) -> {a}  size {a.size()}, capacity {a.capacity()}")
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
add(3) -> [3]  size 1, capacity 2
add(4) -> [3, 4]  size 2, capacity 2
add(5) -> [3, 4, 5]  size 3, capacity 4
add(6) -> [3, 4, 5, 6]  size 4, capacity 4
add(7) -> [3, 4, 5, 6, 7]  size 5, capacity 8
test_bad_capacity (__main__.TestGrowableIntArray.test_bad_capacity) ... ok
test_bad_index (__main__.TestGrowableIntArray.test_bad_index) ... ok
test_values_and_capacity (__main__.TestGrowableIntArray.test_values_and_capacity) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Line by line. The first five output lines show the doubling at work: capacity 2 fills up at the second `add`, becomes 4 at the third, and 8 at the fifth. `Arrays.copyOf(data, 2 * data.length)` creates a new array twice as long, copies the old values in, and fills the rest with zeros; the old array is then garbage. Because the capacity doubles, n additions cause only about $\log_2 n$ copies, and the total copying work stays below $2n$, which is why `ArrayList.add` is fast on average. The checks cover the ordinary values, both ends of the valid range (`get(0)` and `get(4)`), both sides just outside it (`get(5)` and `get(-1)`), and the constructor's own rule. In Python, the method `setUp` runs before *each* test, so every test starts from the same five-element array; this is the `unittest` counterpart of JUnit's `@BeforeEach`.

A tempting wrong solution. `getWrong` tests `index > size` instead of `index >= size`. For `get(5)` on a five-element array, the check passes (5 is not greater than 5), and `data[5]` is read. There is no crash, because the array has capacity 8, so cell 5 exists; it simply holds a 0 that nobody ever

added. The output line `getWrong(5) = 0 <- a value that was never added` is a *silent* wrong answer, the most dangerous kind. Java's own `ArrayIndexOutOfBoundsException` cannot help, because it checks against the capacity, not the size. Only a test at exactly `index == size` catches it.

> **CHECK YOURSELF.** Starting with capacity 1, how many times does the array grow while 100 values are added, and what is the capacity at the end?
>
> *Answer.* It grows when adding the 2nd, 3rd, 5th, 9th, 17th, 33rd and 65th values: 7 times, and at the end the capacity is $2^7 = 128$.

8.7 New Worked Problem: Interfaces and Their Tests

8.7.1 Problem 8.18 — One interface, two implementations, one set of tests

> **PROBLEM.** Profitina Laundry prices an order by weight. The *standard* rule charges Rp7,000 per kilogram; the *express* rule charges Rp12,000 per kilogram plus a flat Rp5,000. Every started kilogram is billed in full (2.3 kg is billed as 3 kg), and a weight of zero or less is an error. Define an interface `PriceRule` with the two rules as implementations, give the interface a method `quote(kg)` that produces a printable quote for any rule, and test both implementations.

In our own words. The rest of the program should be able to hold "some price rule" without knowing which one; that is exactly what an interface is for (Chapter 6). The shared part, rounding up and rejecting bad weights, must be written once.

Design. **Figure 8.7** shows the design as a UML class diagram. The interface declares two abstract methods, `priceFor` and `name`, and one *default method*, `quote`, that is implemented in the interface itself in terms of the other two. Each implementation keeps its rates in `private final` fields.

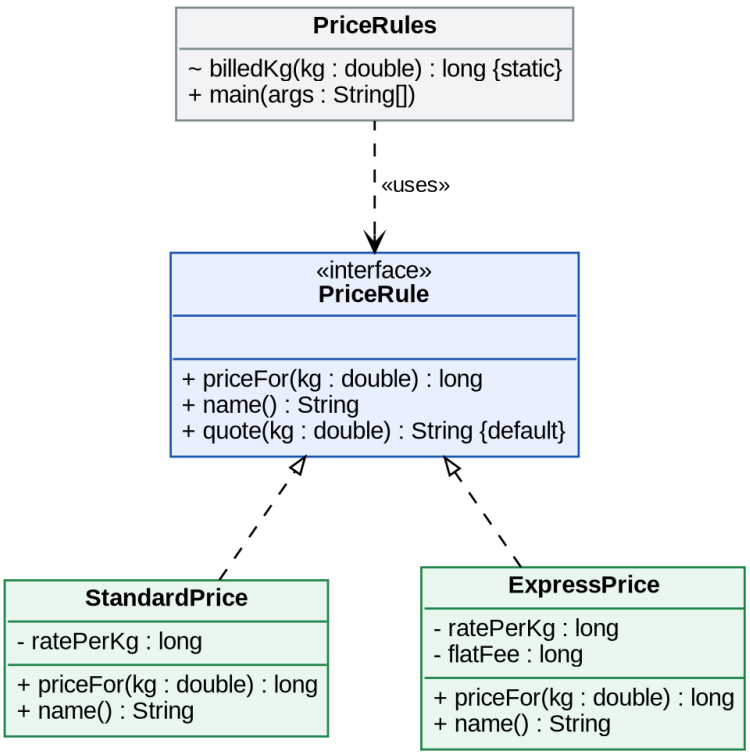


Figure 8.7 UML class diagram of Problem 8.18. The dashed lines with hollow triangles mean "implements"; `quote` is a default method of the interface.

Before writing code, write the expected values by hand. Standard, 2.3 kg: $\lceil 2.3 \rceil = 3$ kg, $3 \times 7000 = 21000$. Express, 2.3 kg: $3 \times 12000 + 5000 = 41000$. Standard, 0.1 kg: $\lceil 0.1 \rceil = 1$ kg, 7000. Here $\lceil x \rceil$, the *ceiling* of x, is the smallest whole number not less than x.

The Java solution. **Listing 8.43** puts the interface, both implementations and the checks in one file.

Listing 8.43 `PriceRules.java`: an interface with a default method, two implementations, and tests that treat both through the interface.

```
// PriceRules.java - Problem 8.18: one interface, two implementations, one set of tests.
public class PriceRules {

    /** How a laundry service turns a weight into a price in rupiah. */
    interface PriceRule {
        /** Price for kg kilograms; every started kilogram is billed in full. */
        long priceFor(double kg);

        String name();

        /** A default method: written once, in terms of the two abstract ones. */
        default String quote(double kg) {
            return name() + " " + kg + " kg = Rp" + priceFor(kg);
        }
    }

    /** Whole kilograms billed, with validation shared by both rules. */
    static long billedKg(double kg) {
        if (kg <= 0) {
            throw new IllegalArgumentException("weight must be positive: " + kg);
        }
        return (long) Math.ceil(kg);              // 2.3 kg is billed as 3 kg
    }

    static final class StandardPrice implements PriceRule {
        private final long ratePerKg;

        StandardPrice(long ratePerKg) {
            this.ratePerKg = ratePerKg;
        }

        @Override
        public long priceFor(double kg) {
            return ratePerKg * billedKg(kg);
        }

        @Override
        public String name() {
            return "standard";
        }
    }

    static final class ExpressPrice implements PriceRule {
        private final long ratePerKg;
        private final long flatFee;
```

Listing 8.43 (Continued) *PriceRules.java: an interface with a default method, two implementations, and tests that treat both through the interface.*

```
        ExpressPrice(long ratePerKg, long flatFee) {
            this.ratePerKg = ratePerKg;
            this.flatFee = flatFee;
        }

        @Override
        public long priceFor(double kg) {
            return ratePerKg * billedKg(kg) + flatFee;
        }

        @Override
        public String name() {
            return "express";
        }
    }

    /** Tempting but wrong: multiply first, then cut off the decimals. */
    static long standardPriceWrong(long ratePerKg, double kg) {
        return (long) (ratePerKg * kg);
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        PriceRule[] rules = {new StandardPrice(7000), new ExpressPrice(12000, 5000)};
        for (PriceRule rule : rules) {              // the loop knows only the interface
            System.out.println(rule.quote(2.3));
        }
        System.out.println("wrong standard 2.3 kg = Rp" + standardPriceWrong(7000, 2.3));
```

Listing 8.43 (Continued) *PriceRules.java: an interface with a default method, two implementations, and tests that treat both through the interface.*

```
        PriceRule standard = rules[0];
        PriceRule express = rules[1];
        check("standard exact kg", 21000L, standard.priceFor(3.0));
        check("standard rounds up", 21000L, standard.priceFor(2.3));
        check("express adds fee", 41000L, express.priceFor(2.3));
        check("smallest bill", 7000L, standard.priceFor(0.1));
        for (PriceRule rule : rules) {
            try {
                rule.priceFor(0);
                throw new AssertionError("no exception");
            } catch (IllegalArgumentException e) {
                check(rule.name() + " rejects 0 kg", "weight must be positive: 0.0",
e.getMessage());
            }
        }
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
standard 2.3 kg = Rp21000
express 2.3 kg = Rp41000
wrong standard 2.3 kg = Rp16099
PASS standard exact kg
PASS standard rounds up
PASS express adds fee
PASS smallest bill
PASS standard rejects 0 kg
PASS express rejects 0 kg
All 6 checks passed.
```

The Python solution. Python's counterpart of a Java interface is an *abstract base class*: a class that inherits from `ABC` and marks methods with `@abstractmethod` cannot be instantiated, and its subclasses must override those methods [8]. **Listing 8.44** also shows a useful testing pattern: a *contract* class, `PriceRuleContract`, holds the tests that every `PriceRule` must pass, and each implementation gets a small test class that mixes the contract in and says how to build its rule.

Listing 8.44 `price_rules.py`: the same design with an abstract base class, and a reusable contract test run against both implementations.

```python
# price_rules.py - Problem 8.18: one interface, two implementations, one set of tests.
import math
import sys
import unittest
from abc import ABC, abstractmethod


class PriceRule(ABC):
    """How a laundry service turns a weight into a price in rupiah."""

    @abstractmethod
    def price_for(self, kg):
        """Price for kg kilograms; every started kilogram is billed in full."""

    @abstractmethod
    def name(self):
        """A short name for printing."""

    def quote(self, kg):
        """A concrete method written once, in terms of the two abstract ones."""
        return f"{self.name()} {kg} kg = Rp{self.price_for(kg)}"


def billed_kg(kg):
    """Whole kilograms billed, with validation shared by both rules."""
    if kg <= 0:
        raise ValueError(f"weight must be positive: {kg}")
    return math.ceil(kg)                    # 2.3 kg is billed as 3 kg


class StandardPrice(PriceRule):
    def __init__(self, rate_per_kg):
        self._rate_per_kg = rate_per_kg

    def price_for(self, kg):
        return self._rate_per_kg * billed_kg(kg)

    def name(self):
        return "standard"


class ExpressPrice(PriceRule):
    def __init__(self, rate_per_kg, flat_fee):
        self._rate_per_kg = rate_per_kg
        self._flat_fee = flat_fee
```

Listing 8.44 (Continued) *price_rules.py: the same design with an abstract base class, and a reusable contract test run against both implementations.*

```
    def price_for(self, kg):
        return self._rate_per_kg * billed_kg(kg) + self._flat_fee

    def name(self):
        return "express"


class PriceRuleContract:
    """Tests every PriceRule must pass; mixed into one TestCase per class."""

    def make_rule(self):
        raise NotImplementedError

    def test_rejects_zero(self):
        with self.assertRaises(ValueError):
            self.make_rule().price_for(0)

    def test_rounds_up(self):
        rule = self.make_rule()
        self.assertEqual(rule.price_for(3.0), rule.price_for(2.3))


class TestStandard(PriceRuleContract, unittest.TestCase):
    def make_rule(self):
        return StandardPrice(7000)

    def test_prices(self):
        self.assertEqual(21000, self.make_rule().price_for(2.3))
        self.assertEqual(7000, self.make_rule().price_for(0.1))


class TestExpress(PriceRuleContract, unittest.TestCase):
    def make_rule(self):
        return ExpressPrice(12000, 5000)

    def test_prices(self):
        self.assertEqual(41000, self.make_rule().price_for(2.3))


class TestAbstract(unittest.TestCase):
    def test_cannot_instantiate_interface(self):
        with self.assertRaises(TypeError):
            PriceRule()
```

Listing 8.44 (Continued) *price_rules.py: the same design with an abstract base class, and a reusable contract test run against both implementations.*

```
if __name__ == "__main__":
    for rule in [StandardPrice(7000), ExpressPrice(12000, 5000)]:
        print(rule.quote(2.3))
    print("wrong standard 2.3 kg = Rp" + str(int(7000 * 2.3)))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
standard 2.3 kg = Rp21000
express 2.3 kg = Rp41000
wrong standard 2.3 kg = Rp16099
test_cannot_instantiate_interface (__main__.TestAbstract.test_cannot_instantiate_interface)
... ok
test_prices (__main__.TestExpress.test_prices) ... ok
test_rejects_zero (__main__.TestExpress.test_rejects_zero) ... ok
test_rounds_up (__main__.TestExpress.test_rounds_up) ... ok
test_prices (__main__.TestStandard.test_prices) ... ok
test_rejects_zero (__main__.TestStandard.test_rejects_zero) ... ok
test_rounds_up (__main__.TestStandard.test_rounds_up) ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.000s

OK
```

Line by line. In `main`, the array has type `PriceRule[]`, and the loop variable has type `PriceRule`: the loop does not know which class it is talking to, and `rule.quote(2.3)` still calls the right `priceFor`, because the object decides, not the variable. The default method `quote` is written once for every present and future rule. The helper `billedKg` holds the two shared rules, rounding up with `Math.ceil` and rejecting non-positive weights, so neither implementation repeats them. Money is kept in `long` rupiah, never in `double`; whole numbers are exact. The Python run shows seven tests from four classes: the two contract tests run once for each implementation, which is exactly the point of the pattern, and `test_cannot_instantiate_interface` confirms that `PriceRule()` itself raises `TypeError`.

A tempting wrong solution. `standardPriceWrong` computes `(long) (ratePerKg * kg)`. It has two bugs, and the output shows their sum: `Rp16099`. The first is a design bug: it bills 2.3 kg instead of 3 kg. The second is a floating-point bug: 7000×2.3 is not exactly 16100 in binary floating point, it is 16099.999999999998, and the cast to `long` *truncates* toward zero, dropping to 16099. The rule is to convert weights to whole billing units first (`Math.ceil`), and only then multiply by an integer rate.

8.7.2 The same tests in JUnit 6

The `check` helper of this chapter does, in miniature, what JUnit 6 does [9]. **Table 8.19** gives the correspondence. The chapter's folder contains JUnit versions of two of the test drivers, in a subfolder kept under the name `junit5` because their API dates from JUnit 5: `junit5/PriceRulesTest.java` and `junit5/GrowableIntArrayTest.java`; each check has become a method marked `@Test`.

Table 8.19 How the self-checking programs of this chapter map onto JUnit 6 and `unittest`.

This chapter (plain Java)	JUnit 6	Python `unittest`
`check("name", expected, actual)`	`assertEquals(expected, actual)` in a `@Test` method	`self.assertEqual(expected, actual)`
`try { ...; throw new AssertionError(); } catch (X e) {...}`	`X e = assertThrows(X.class, () -> ...)`	`with self.assertRaises(X):`
code at the start of `main`	a method marked `@BeforeEach`	`setUp(self)`
`All N checks passed.`	the runner's summary	`Ran N tests ... OK`

To run them you need the JUnit 6 "console launcher", a single file named `junit-platform-console-standalone-<version>.jar`, downloaded once from the JUnit web site or from Maven Central. From the chapter's `java` folder:

```
javac -d out -cp junit-platform-console-standalone.jar *.java junit5/*.java
java -jar junit-platform-console-standalone.jar execute --class-path out --scan-class-path
```

The first command compiles the programs and the tests into the folder `out`; the second finds every `@Test` method and runs it, printing a tree of results and a summary. In VS Code, the Extension Pack for Java (Chapter 1) recognizes `@Test` methods once the JAR is on the project's class path, and shows a run button beside each one. The launcher could not be downloaded on the machine that built this book, so the two JUnit files were compiled against a minimal stand-in for the JUnit annotations, and a small runner reported all eight `@Test` methods passing. That is not real JUnit, so no JUnit output is quoted here; the commands above show JUnit's own report.

> **CHECK YOURSELF.** Add a third rule, `IronOnlyPrice`, Rp5,000 per kilogram with a minimum bill of Rp10,000. What does it charge for 1.2 kg and for 2.5 kg, and which methods must the class define?
>
> *Answer.* It defines `priceFor` and `name`; `quote` comes free from the interface. 1.2 kg: $\lceil 1.2 \rceil = 2$, $2 \times 5000 = 10000$, which equals the minimum: Rp10,000. 2.5 kg: $3 \times 5000 = 15000$. In Python, adding `class TestIronOnly(PriceRuleContract, unittest.TestCase)` runs the whole contract against it for free.

8.8 New Worked Problems: Inheritance, Polymorphism, and Access Control

8.8.1 Problem 8.19 — A class hierarchy traced by hand

> **PROBLEM.** Profitina Laundry has three services: `Wash` (Rp7,000 per kg), `Iron` (Rp5,000 per kg) and `Express`, a special wash at twice the wash rate whose description adds "(same day)". They share an abstract superclass `Service` with a final method `price(kg)` and a method `describe()`. There are also two *overloaded* static methods `label(Service)` and `label(Wash)`. Before running **Listing 8.45**, predict by hand every line it prints, and explain each prediction.

In our own words. The problem is about the two different moments at which Java chooses which method to run. *Overloading* (Chapter 6) is resolved by the compiler, from the *declared* types of the arguments. *Overriding* (Chapter 7) is resolved while the program runs, from the *actual* class of the object. Mixing them up is the most common misunderstanding of polymorphism [10].

Design. **Figure 8.8** shows the hierarchy. `price` is `final` in `Service`, so no subclass can change it; it calls the abstract `pricePerKg`, which each subclass provides. A method in the superclass that fixes the order of steps and lets subclasses fill in one step is the *template method* pattern [11].

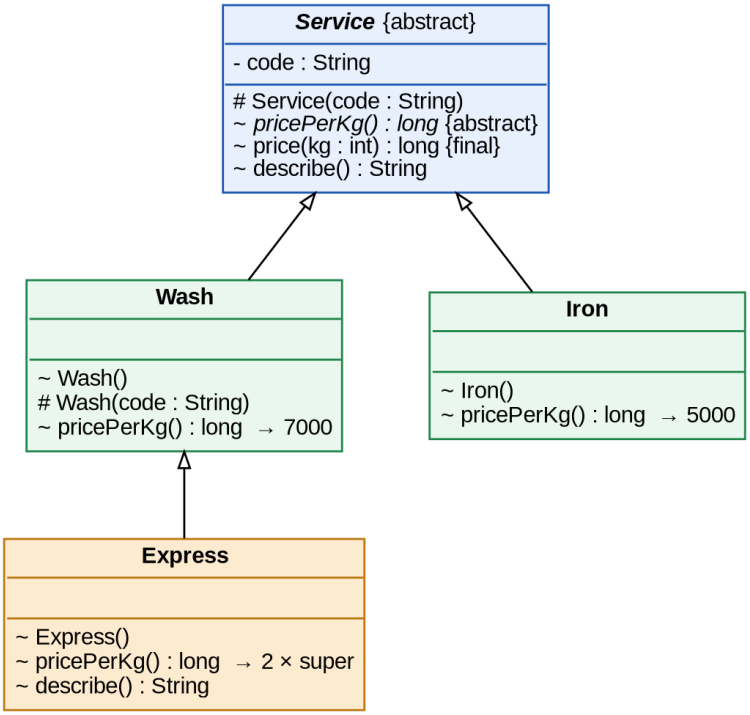


Figure 8.8 The service hierarchy of Problem 8.19. `Express` extends `Wash`, overrides `pricePerKg` and `describe`, and uses `super` to reach the `Wash` versions.

Listing 8.45 `ServiceDispatch.java`: an abstract class, three subclasses, overriding with `super`, and two overloads of `label`.

```java
// ServiceDispatch.java - Problem 8.19: a class hierarchy traced by hand.
public class ServiceDispatch {

    abstract static class Service {
        private final String code;

        protected Service(String code) {
            this.code = code;
        }

        abstract long pricePerKg();

        /** The same code for every service; pricePerKg() is chosen at run time. */
        final long price(int kg) {
            return pricePerKg() * kg;
        }

        String describe() {
            return code + " @ " + pricePerKg() + "/kg";
        }
    }

    static class Wash extends Service {
        Wash() {
            this("WASH");
        }

        protected Wash(String code) {           // lets a subclass choose its code
            super(code);
        }

        @Override
        long pricePerKg() {
            return 7000;
        }
    }

    static class Iron extends Service {
        Iron() {
            super("IRON");
        }
```

Listing 8.45 (Continued) *ServiceDispatch.java: an abstract class, three subclasses, overriding with super, and two overloads of label.*

```
        @Override
        long pricePerKg() {
            return 5000;
        }
    }

    static class Express extends Wash {
        Express() {
            super("EXPRESS");
        }

        @Override
        long pricePerKg() {
            return 2 * super.pricePerKg();      // twice the Wash rate
        }

        @Override
        String describe() {
            return super.describe() + " (same day)";
        }
    }

    // Two OVERLOADS: the compiler picks one from the declared type of the argument.
    static String label(Service s) {
        return "service: " + s.describe();
    }

    static String label(Wash w) {
        return "washing: " + w.describe();
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }
```

Listing 8.45 (Continued) *ServiceDispatch.java: an abstract class, three subclasses, overriding with super, and two overloads of label.*

```
    public static void main(String[] args) {
        Service[] services = {new Wash(), new Iron(), new Express()};
        for (Service s : services) {                          // line A
            System.out.println(s.describe() + ", 3 kg = " + s.price(3));
        }
        Service s = new Express();                            // line B
        Wash w = new Express();                               // line C
        System.out.println(label(s));                         // line D
        System.out.println(label(w));                         // line E
        System.out.println(label((Wash) s));                  // line F

        check("express price", 42000L, services[2].price(3));
        check("overload by declared type", "service: EXPRESS @ 14000/kg (same day)",
label(s));
        check("cast changes the overload", "washing: EXPRESS @ 14000/kg (same day)",
label((Wash) s));
        check("express is a wash", true, s instanceof Wash);
        System.out.println("All " + passed + " checks passed.");
    }
}
```

The trace by hand. **Table 8.20** is the prediction, made before running. For each call, the second column is the *declared* type of the variable, known to the compiler; the third is the class of the *object*, known only at run time.

Table 8.20 Hand trace of `ServiceDispatch.main`. Overloads are chosen from the declared type; overridden methods from the object's class.

Line	Declared type	Object's class	Which `label` (compiler)	Which `describe`/`pricePerKg` (run time)	Prints
A, 1st	`Service`	`Wash`	—	`Service.describe`, `Wash.pricePerKg`	`WASH @ 7000/kg, 3 kg = 21000`
A, 2nd	`Service`	`Iron`	—	`Service.describe`, `Iron.pricePerKg`	`IRON @ 5000/kg, 3 kg = 15000`
A, 3rd	`Service`	`Express`	—	`Express.describe`, `Express.pricePerKg`	`EXPRESS @ 14000/kg (same day), 3 kg = 42000`
D	`Service`	`Express`	`label(Service)`	`Express.describe`	`service: EXPRESS @ 14000/kg (same day)`
E	`Wash`	`Express`	`label(Wash)`	`Express.describe`	`washing: EXPRESS @ 14000/kg (same day)`
F	`Wash` (cast)	`Express`	`label(Wash)`	`Express.describe`	`washing: EXPRESS @ 14000/kg (same day)`

Now run it and compare:

```
WASH @ 7000/kg, 3 kg = 21000
IRON @ 5000/kg, 3 kg = 15000
EXPRESS @ 14000/kg (same day), 3 kg = 42000
service: EXPRESS @ 14000/kg (same day)
washing: EXPRESS @ 14000/kg (same day)
washing: EXPRESS @ 14000/kg (same day)
PASS express price
PASS overload by declared type
PASS cast changes the overload
PASS express is a wash
All 4 checks passed.
```

Line by line through the trace. In the third pass of loop A, `s.describe()` runs `Express.describe`, which calls `super.describe()`, the version in `Service`. Inside it, `pricePerKg()` is called on the same object, which is still an `Express`, so `Express.pricePerKg` runs; it calls `super.pricePerKg()`, the `Wash` version, gets 7000 and doubles it. Dynamic dispatch applies to *every* call on the object, including calls made from inside superclass code. Line D is the lesson of the problem: `s` holds an `Express`, which *is* a `Wash`, yet `label(Service)` is chosen, because the compiler sees only `Service s` and must decide before the program runs. Inside `label`, `s.describe()` is again dispatched at run time, so the text is the `Express` description. Line F shows that a cast changes the declared type and therefore the overload. The `Wash(String code)` constructor is `protected`: `Express` needs it to pass its own code up, but ordinary users should create a wash with `new Wash()`.

The Python version. Python has no overloading by declared type (a second `def label` would simply replace the first) and no declared types to go by. The standard library's `functools.singledispatch` [8] offers something related but different: it chooses the function by the *run-time* class of the first argument. **Listing 8.46** therefore prints `washing:` for the `Express` object, which differs from Java's line D; that difference is the whole point.

Listing 8.46 `service_dispatch.py`: the same hierarchy in Python, with `singledispatch` choosing by run-time class.

```python
# service_dispatch.py - Problem 8.19: a class hierarchy traced by hand.
import sys
import unittest
from abc import ABC, abstractmethod
from functools import singledispatch


class Service(ABC):
    def __init__(self, code):
        self._code = code

    @abstractmethod
    def price_per_kg(self):
        """Rupiah per kilogram."""

    def price(self, kg):
        """The same code for every service; price_per_kg() is chosen at run time."""
        return self.price_per_kg() * kg

    def describe(self):
        return f"{self._code} @ {self.price_per_kg()}/kg"


class Wash(Service):
    def __init__(self, code="WASH"):
        super().__init__(code)

    def price_per_kg(self):
        return 7000


class Iron(Service):
    def __init__(self):
        super().__init__("IRON")

    def price_per_kg(self):
        return 5000


class Express(Wash):
    def __init__(self):
        super().__init__("EXPRESS")

    def price_per_kg(self):
        return 2 * super().price_per_kg()   # twice the Wash rate
```

Listing 8.46 (Continued) *service_dispatch.py: the same hierarchy in Python, with singledispatch choosing by run-time class.*

```
    def describe(self):
        return super().describe() + " (same day)"


# Python has no overloading by declared type. singledispatch chooses
# by the RUN-TIME type of the argument instead.
@singledispatch
def label(s):
    return "service: " + s.describe()


@label.register
def _(w: Wash):
    return "washing: " + w.describe()


class TestDispatch(unittest.TestCase):
    def test_prices(self):
        self.assertEqual([21000, 15000, 42000],
                         [s.price(3) for s in (Wash(), Iron(), Express())])

    def test_label_uses_runtime_type(self):
        s = Express()
        self.assertEqual("washing: EXPRESS @ 14000/kg (same day)", label(s))

    def test_label_for_plain_service(self):
        self.assertEqual("service: IRON @ 5000/kg", label(Iron()))


if __name__ == "__main__":
    for s in [Wash(), Iron(), Express()]:
        print(f"{s.describe()}, 3 kg = {s.price(3)}")
    s = Express()
    print(label(s))
    print(label(Iron()))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
WASH @ 7000/kg, 3 kg = 21000
IRON @ 5000/kg, 3 kg = 15000
EXPRESS @ 14000/kg (same day), 3 kg = 42000
washing: EXPRESS @ 14000/kg (same day)
service: IRON @ 5000/kg
test_label_for_plain_service (__main__.TestDispatch.test_label_for_plain_service) ... ok
test_label_uses_runtime_type (__main__.TestDispatch.test_label_uses_runtime_type) ... ok
test_prices (__main__.TestDispatch.test_prices) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

A tempting wrong solution. Two, both about `super`. The tempting wrong *prediction* is "line D prints `washing:`, because the object is an `Express`, which is a `Wash`": that is the run-time reasoning applied to a compile-time decision. The tempting wrong *code* is to write `return 2 * pricePerKg();` in `Express`, forgetting `super.`. The call then refers to `Express.pricePerKg` itself, which calls itself again, forever, until Java throws `StackOverflowError` (Python: `RecursionError`). `super.m()` means "the version of `m` in my superclass"; plain `m()` means "the version for this object", which inside an override is the override itself.

CHECK YOURSELF. Suppose `Iron` also extended `Wash` instead of `Service`, keeping its own `pricePerKg` and constructor code "IRON" through the protected constructor. What would `label(new Iron())` print, with the argument's declared type being `Iron`?

Answer. The compiler chooses the most specific applicable overload. `Iron` is now a subclass of `Wash`, so both `label(Service)` and `label(Wash)` apply, and `label(Wash)` is more specific: it prints `washing: IRON @ 5000/kg`.

8.8.2 Problem 8.20 — Refactoring two duplicated classes into a hierarchy

PROBLEM. A small game has two classes, `Player` and `Enemy`. Both have the private fields `name`, `health`, `power` and `defense`, the same getters, a public `setHealth`, a method `attack` (the player's takes an `Enemy`, the enemy's takes a `Player`) and an identical method `takeDamage(damage)` that subtracts `damage - defense` from `health` if that is positive and prints "`<name> died!`" when health reaches zero or less. A test program creates a hero (health 100, power 25, defense 10) and a villain (80, 20, 5) and makes three attacks: hero, villain, hero. Explain what the program prints, identify the design problems, and refactor the two classes into a hierarchy in which a player can also heal with a potion and an enemy fights harder when badly hurt.

In our own words. Two classes that are copies of each other are a maintenance trap: a rule changed in one and forgotten in the other makes the game unfair in a way nobody notices. The refactoring called *extract superclass* [12] moves the common part into one abstract class. After that, each subclass contains only what makes it different.

Design. First, the trace of the original program by hand. Hero attacks: the villain takes $25 - 5 = 20$, health $80 \to 60$. Villain attacks: the hero takes $20 - 10 = 10$, health $100 \to 90$. Hero attacks: $60 \to 40$. Nobody reaches zero, and `takeDamage` prints only on death, so **the original program prints nothing at all**. Running it confirms this: it ends without a single line of output. A test that prints nothing and checks nothing tells us nothing, which is the first problem. **Table 8.21** lists the others, each with its fix.

Table 8.21 The design problems of the duplicated `Player` and `Enemy` classes and how the refactoring fixes each.

Problem in the original	Chapter	Fix in `FighterArena`
Two identical copies of fields and methods	7	abstract superclass `Fighter` holds them once
`attack(Enemy)` and `attack(Player)`: two overloads for one idea	6, 7	one `attack(Fighter target)`, polymorphic
`public setHealth` lets anyone set any health	7	no setter; health changes only through `takeDamage` and `restore`
Health can go below zero	5	`Math.max(0, ...)` keeps it in $[0, maxHealth]$
`takeDamage` prints to the console	6	methods return values; `main` decides what to print
No validation of the starting stats	5	the constructor throws `IllegalArgumentException`
A test program with no checks	6	`check` calls with the hand-traced values

Figure 8.9 shows the classes before and after. The protected *hook* method `attackPower()` is the one place where subclasses may change the fighting rule; the `final` method `attack` guarantees that every attack follows the same steps.

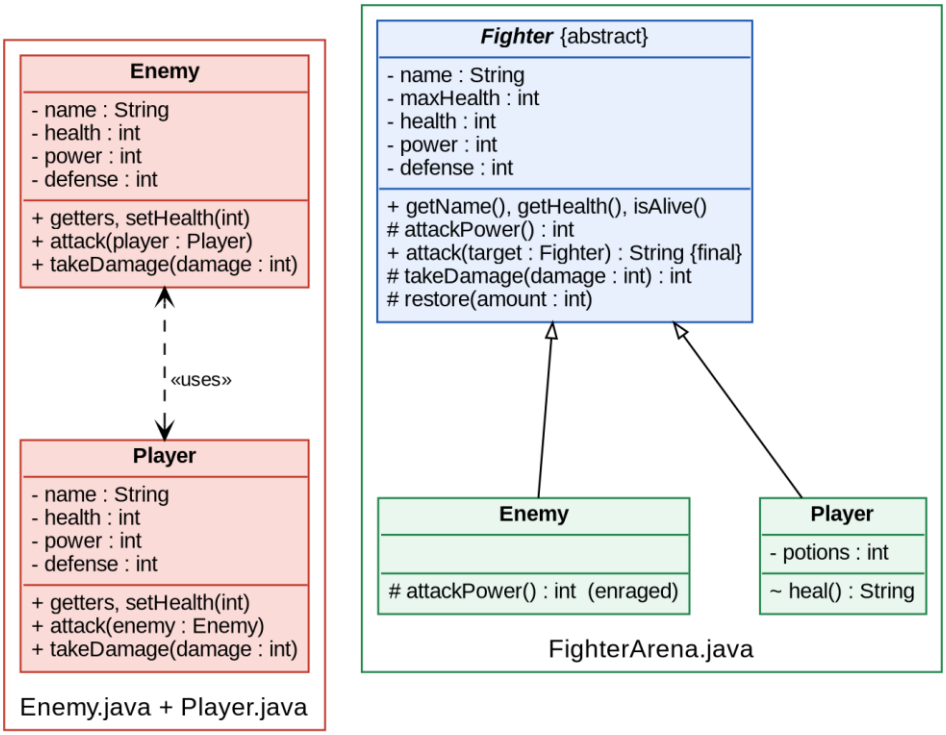


Figure 8.9 Before (left): two copies of the same fields and methods, each class using the other. After (right): one abstract `Fighter`, and subclasses that add only what differs.

Listing 8.47 and **Listing 8.48** implement the design of **Figure 8.9** and replay the three blows of the original test program.

Listing 8.47 `FighterArena.java`: the refactored hierarchy with a final template method, a protected hook and no public setter.

```java
// FighterArena.java - Problem 8.20: two duplicated classes refactored into a hierarchy.
public class FighterArena {

    /** Everything Player and Enemy used to repeat, written once. */
    abstract static class Fighter {
        private final String name;
        private final int maxHealth;
        private int health;                    // changed only by takeDamage/heal
        private final int power;
        private final int defense;

        protected Fighter(String name, int health, int power, int defense) {
            if (health <= 0 || power < 0 || defense < 0) {
                throw new IllegalArgumentException("bad stats for " + name);
            }
            this.name = name;
            this.maxHealth = health;
            this.health = health;
            this.power = power;
            this.defense = defense;
        }

        public String getName() {
            return name;
        }

        public int getHealth() {
            return health;
        }

        public boolean isAlive() {
            return health > 0;
        }

        protected int getPower() {
            return power;
        }

        protected int getMaxHealth() {
            return maxHealth;
        }
```

Listing 8.47 (Continued) *FighterArena.java: the refactored hierarchy with a final template method, a protected hook and no public setter.*

```
        /** How hard I hit right now. Subclasses may change this (a hook). */
        protected int attackPower() {
            return power;
        }

        /** The same for every fighter, so it is final: subclasses cannot change it. */
        public final String attack(Fighter target) {
            int dealt = target.takeDamage(attackPower());
            return name + " hits " + target.getName() + " for " + dealt
                    + " -> " + target.getName() + " has " + target.getHealth()
                    + (target.isAlive() ? "" : " (defeated)");
        }

        /** Defense absorbs part of the blow; health never drops below 0. */
        protected int takeDamage(int damage) {
            int reduced = Math.max(0, damage - defense);
            health = Math.max(0, health - reduced);
            return reduced;
        }

        protected void restore(int amount) {
            health = Math.min(maxHealth, health + amount);
        }
    }

    static final class Player extends Fighter {
        private int potions;

        Player(String name, int health, int power, int defense, int potions) {
            super(name, health, power, defense);
            this.potions = potions;
        }

        /** Drinks a potion (+30 health, never above the maximum) if one is left. */
        String heal() {
            if (potions == 0) {
                return getName() + " has no potions left";
            }
            potions--;
            restore(30);
            return getName() + " drinks a potion -> " + getHealth();
        }
    }
```

Listing 8.47 (Continued) *FighterArena.java: the refactored hierarchy with a final template method, a protected hook and no public setter.*

```
    static final class Enemy extends Fighter {
        Enemy(String name, int health, int power, int defense) {
            super(name, health, power, defense);
        }

        /** Below half health an enemy is enraged and hits 10 harder. */
        @Override
        protected int attackPower() {
            boolean enraged = getHealth() < getMaxHealth() / 2;
            return enraged ? getPower() + 10 : getPower();
        }
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        Player hero = new Player("Hero", 100, 25, 10, 1);
        Enemy villain = new Enemy("Villain", 80, 20, 5);

        // The three blows of EnemyPlayerTest.java:
        System.out.println(hero.attack(villain));
        System.out.println(villain.attack(hero));
        System.out.println(hero.attack(villain));
        check("villain after two blows", 40, villain.getHealth());
        check("hero after one blow", 90, hero.getHealth());

        // Fight on: the two fighters take turns until one is defeated.
        Fighter[] turn = {villain, hero};      // villain moves next
        int i = 0;
        while (hero.isAlive() && villain.isAlive()) {
            Fighter attacker = turn[i % 2];
            Fighter defender = turn[(i + 1) % 2];
            System.out.println(attacker.attack(defender));
            i++;
        }
        System.out.println(hero.heal());
        System.out.println(hero.heal());
        check("villain defeated", false, villain.isAlive());
```

Listing 8.47 (Continued) *FighterArena.java: the refactored hierarchy with a final template method, a protected hook and no public setter.*

```
        check("health never negative", 0, villain.getHealth());
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
Hero hits Villain for 20 -> Villain has 60
Villain hits Hero for 10 -> Hero has 90
Hero hits Villain for 20 -> Villain has 40
PASS villain after two blows
PASS hero after one blow
Villain hits Hero for 10 -> Hero has 80
Hero hits Villain for 20 -> Villain has 20
Villain hits Hero for 20 -> Hero has 60
Hero hits Villain for 20 -> Villain has 0 (defeated)
Hero drinks a potion -> 90
Hero has no potions left
PASS villain defeated
PASS health never negative
All 4 checks passed.
```

Listing 8.48 `fighter_arena.py`: the same hierarchy in Python, with a read-only `health` property.

```python
# fighter_arena.py - Problem 8.20: two duplicated classes refactored into a hierarchy.
import sys
import unittest
from abc import ABC


class Fighter(ABC):
    """Everything Player and Enemy used to repeat, written once."""

    def __init__(self, name, health, power, defense):
        if health <= 0 or power < 0 or defense < 0:
            raise ValueError(f"bad stats for {name}")
        self._name = name
        self._max_health = health
        self._health = health               # changed only by take_damage/restore
        self._power = power
        self._defense = defense

    @property
    def name(self):
        return self._name

    @property
    def health(self):                       # read-only: there is no setter
        return self._health

    def is_alive(self):
        return self._health > 0

    def attack_power(self):
        """How hard I hit right now. Subclasses may change this (a hook)."""
        return self._power

    def attack(self, target):
        """The same for every fighter; by convention, subclasses leave it alone."""
        dealt = target._take_damage(self.attack_power())
        return (f"{self._name} hits {target.name} for {dealt} -> {target.name} has "
                f"{target.health}" + ("" if target.is_alive() else " (defeated)"))

    def _take_damage(self, damage):
        """Defense absorbs part of the blow; health never drops below 0."""
        reduced = max(0, damage - self._defense)
        self._health = max(0, self._health - reduced)
        return reduced
```

Listing 8.48 (Continued) *fighter_arena.py: the same hierarchy in Python, with a read-only health property.*

```
    def _restore(self, amount):
        self._health = min(self._max_health, self._health + amount)


class Player(Fighter):
    def __init__(self, name, health, power, defense, potions):
        super().__init__(name, health, power, defense)
        self._potions = potions

    def heal(self):
        """Drinks a potion (+30 health, never above the maximum) if one is left."""
        if self._potions == 0:
            return f"{self._name} has no potions left"
        self._potions -= 1
        self._restore(30)
        return f"{self._name} drinks a potion -> {self._health}"


class Enemy(Fighter):
    def attack_power(self):
        """Below half health an enemy is enraged and hits 10 harder."""
        enraged = self._health < self._max_health // 2
        return self._power + 10 if enraged else self._power


class TestArena(unittest.TestCase):
    def test_three_blows(self):
        hero, villain = Player("Hero", 100, 25, 10, 1), Enemy("Villain", 80, 20, 5)
        hero.attack(villain)
        villain.attack(hero)
        hero.attack(villain)
        self.assertEqual((90, 40), (hero.health, villain.health))

    def test_health_is_read_only(self):
        with self.assertRaises(AttributeError):
            Player("Hero", 100, 25, 10, 1).health = -500

    def test_bad_stats(self):
        with self.assertRaises(ValueError):
            Enemy("Ghost", 0, 1, 1)
```

Listing 8.48 (Continued) *fighter_arena.py: the same hierarchy in Python, with a read-only health property.*

```
if __name__ == "__main__":
    hero = Player("Hero", 100, 25, 10, 1)
    villain = Enemy("Villain", 80, 20, 5)
    print(hero.attack(villain))
    print(villain.attack(hero))
    print(hero.attack(villain))
    turn = [villain, hero]                  # villain moves next
    i = 0
    while hero.is_alive() and villain.is_alive():
        print(turn[i % 2].attack(turn[(i + 1) % 2]))
        i += 1
    print(hero.heal())
    print(hero.heal())
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
Hero hits Villain for 20 -> Villain has 60
Villain hits Hero for 10 -> Hero has 90
Hero hits Villain for 20 -> Villain has 40
Villain hits Hero for 10 -> Hero has 80
Hero hits Villain for 20 -> Villain has 20
Villain hits Hero for 20 -> Hero has 60
Hero hits Villain for 20 -> Villain has 0 (defeated)
Hero drinks a potion -> 90
Hero has no potions left
test_bad_stats (__main__.TestArena.test_bad_stats) ... ok
test_health_is_read_only (__main__.TestArena.test_health_is_read_only) ... ok
test_three_blows (__main__.TestArena.test_three_blows) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Line by line. The first three output lines reproduce the hand trace exactly (60, 90, 40), and the two checks after them lock those numbers in. Then the fight continues with the two fighters in an array of type `Fighter[]`; the loop calls `attacker.attack(defender)` without knowing which is which. Follow the villain's power: at health 40 it is not enraged ($40 < 80/2$ is false), so it hits for $20 - 10 = 10$; at health 20 it *is* enraged, `attackPower()` returns 30, and the hero loses $30 - 10 = 20$. The enemy's special behavior lives entirely in its override of the hook; `attack` itself, being `final`, cannot be changed by any subclass. After the villain is defeated, the player drinks the one potion ($60 + 30 = 90$) and the second `heal()` reports that none are left.

Access control is worth reading closely. `health` is `private` in `Fighter`; subclasses cannot touch it directly, so even `Player.heal()` must go through the `protected` method `restore`, which enforces the maximum. `takeDamage` is `protected`: the `attack` method of one fighter calls it on *another* fighter, which is allowed because the call is made from code inside `Fighter`. One honest caveat: all these classes are nested inside one file, and Java lets code in the same top-level class see even `private` members of the nested classes. Access levels show their full strength when classes live in separate files and packages (Chapter 7); the design here is written as if they did. In Python, `health` is a read-only `property`, so `hero.health = -500` raises `AttributeError`, as the test `test_health_is_read_only` shows.

A tempting wrong solution. The quickest "refactoring" is to make `Enemy extends Player`, since the code is the same. It removes the duplication and compiles, but it claims that every enemy *is a* player: an enemy would inherit the potions and `heal()`, and any method accepting a `Player` would accept an enemy. Inheritance expresses a true "is a special kind of" relationship; when two classes are siblings, the shared part belongs in a common superclass, not in one of them.

> **CHECK YOURSELF.** Add a `Boss` that extends `Enemy` and ignores the first 15 points of every blow instead of its defense. Which method does it override, and what happens to the hero's attack of power 25 against a boss with defense 5?
>
> *Answer.* `Boss` overrides the protected method `takeDamage`. Because `health` is private in `Fighter`, the override cannot subtract from it directly; the clean solution adds a protected helper `loseHealth(int amount)` to `Fighter`, the mirror image of `restore`, and `Boss.takeDamage` computes `Math.max(0, damage - 15)`, calls `loseHealth` and returns the amount. The hero's blow does $25 - 15 = 10$ damage.

8.8.3 Problem 8.21 — An encapsulation bug hunt

> **PROBLEM.** The class `LeakyOrder` in **Listing 8.49** compiles and runs, yet code outside the class can put an order into an impossible state in four different ways. Find the four bugs, demonstrate each one, and write `SafeOrder`, which has the same data but cannot be broken from outside.

In our own words. *Encapsulation* (Chapter 7) means that an object alone controls its own state, so the rules about that state can be enforced in one place. A bug in encapsulation is any path by which outside code changes the state without passing through a check.

Design. Read the class slowly and ask of every field: "who can change this?" The four answers are the four bugs (**Table 8.22**). Bugs 2 and 4 are both *aliasing*: two references to one mutable object, one of them held by a stranger. **Figure 8.10** draws it.

Table 8.22 The four encapsulation bugs of `LeakyOrder`. Every one is a path to the state that bypasses a check.

Bug	Where	How outside code breaks the order	Fix
1	`public String customer`	`leaky.customer = null`	`private final` field, getter only
2	constructor stores `items`	the caller keeps `bag` and changes it later	copy the list *in*: `new ArrayList<>(items)`
3	`setKg` without checks	`setKg(-2)`	validate, throw `IllegalArgumentException`
4	`getItems()` returns the field	`getItems().clear()`	return a read-only view *out*

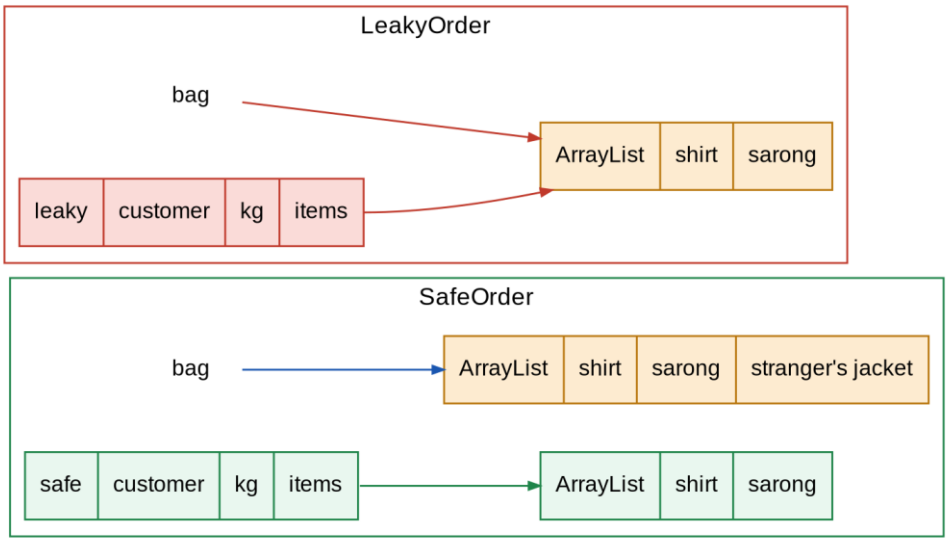


Figure 8.10 Aliasing. Top: `LeakyOrder` and the caller share one list, so either can change it. Bottom: `SafeOrder` copied the list, so the caller's later changes do not reach the order.

Listing 8.49 `OrderBugHunt.java`: a class with four encapsulation bugs, demonstrated, and the fixed class, tested.

```java
// OrderBugHunt.java - Problem 8.21: find four encapsulation bugs, then fix them.
import java.util.ArrayList;
import java.util.Collections;

public class OrderBugHunt {

    /** The class as first written. It compiles, and it is broken in four ways. */
    static class LeakyOrder {
        public String customer;                              // bug 1: public field
        private double kg;
        private final ArrayList<String> items;

        LeakyOrder(String customer, double kg, ArrayList<String> items) {
            this.customer = customer;
            this.kg = kg;
            this.items = items;                              // bug 2: stores the caller's
list
        }

        void setKg(double kg) {
            this.kg = kg;                                    // bug 3: no validation
        }

        ArrayList<String> getItems() {
            return items;                                    // bug 4: hands out the inside
        }

        double getKg() {
            return kg;
        }
    }

    /** The same order with every bug fixed. */
    static final class SafeOrder {
        private final String customer;
        private double kg;
        private final ArrayList<String> items;

        SafeOrder(String customer, double kg, ArrayList<String> items) {
            if (customer == null || customer.isBlank()) {
                throw new IllegalArgumentException("customer is required");
            }
            this.customer = customer;
            setKg(kg);                                       // one place checks weights
            this.items = new ArrayList<>(items);             // defensive copy IN
        }
```

Listing 8.49 (Continued) *OrderBugHunt.java: a class with four encapsulation bugs, demonstrated, and the fixed class, tested.*

```
        void setKg(double kg) {
            if (kg <= 0 || kg > 50) {
                throw new IllegalArgumentException("weight out of range: " + kg);
            }
            this.kg = kg;
        }

        java.util.List<String> getItems() {
            return Collections.unmodifiableList(items);      // read-only view OUT
        }

        String getCustomer() {
            return customer;
        }

        double getKg() {
            return kg;
        }
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        ArrayList<String> bag = new ArrayList<>();
        bag.add("shirt");
        bag.add("sarong");

        LeakyOrder leaky = new LeakyOrder("Sari", 3.5, bag);
        leaky.customer = null;                                  // bug 1 in action
        bag.add("stranger's jacket");                           // bug 2 in action
        leaky.setKg(-2);                                        // bug 3 in action
        leaky.getItems().clear();                               // bug 4 in action
        System.out.println("leaky: customer=" + leaky.customer + ", kg=" + leaky.getKg()
                + ", items=" + leaky.getItems() + ", caller's bag=" + bag);
```

Listing 8.49 (Continued) *OrderBugHunt.java: a class with four encapsulation bugs, demonstrated, and the fixed class, tested.*

```
        bag = new ArrayList<>();
        bag.add("shirt");
        bag.add("sarong");
        SafeOrder safe = new SafeOrder("Sari", 3.5, bag);
        bag.add("stranger's jacket");
        try {
            safe.setKg(-2);
        } catch (IllegalArgumentException e) {
            System.out.println("setKg(-2) refused: " + e.getMessage());
        }
        try {
            safe.getItems().clear();
        } catch (UnsupportedOperationException e) {
            System.out.println("getItems().clear() refused:
UnsupportedOperationException");
        }
        System.out.println("safe : customer=" + safe.getCustomer() + ", kg=" + safe.getKg()
                + ", items=" + safe.getItems());

        check("items unaffected by caller", "[shirt, sarong]", safe.getItems().toString());
        check("weight kept", 3.5, safe.getKg());
        try {
            new SafeOrder(" ", 1, new ArrayList<>());
            throw new AssertionError("no exception");
        } catch (IllegalArgumentException e) {
            check("blank customer refused", "customer is required", e.getMessage());
        }
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
leaky: customer=null, kg=-2.0, items=[], caller's bag=[]
setKg(-2) refused: weight out of range: -2.0
getItems().clear() refused: UnsupportedOperationException
safe : customer=Sari, kg=3.5, items=[shirt, sarong]
PASS items unaffected by caller
PASS weight kept
PASS blank customer refused
All 3 checks passed.
```

Listing 8.50 `order_bug_hunt.py`: the same hunt in Python, fixed with properties, a validating setter and tuples.

```python
# order_bug_hunt.py - Problem 8.21: find four encapsulation bugs, then fix them.
import sys
import unittest


class LeakyOrder:
    """The class as first written. It runs, and it is broken in four ways."""

    def __init__(self, customer, kg, items):
        self.customer = customer        # bug 1: a public attribute, no checks
        self.kg = kg                    # bug 3: anyone may assign any weight
        self.items = items              # bug 2 and 4: the caller's list, shared


class SafeOrder:
    """The same order with every bug fixed."""

    def __init__(self, customer, kg, items):
        if not customer or not customer.strip():
            raise ValueError("customer is required")
        self._customer = customer
        self.kg = kg                    # goes through the property setter below
        self._items = list(items)       # defensive copy IN

    @property
    def customer(self):                 # read-only: no setter is defined
        return self._customer

    @property
    def kg(self):
        return self._kg

    @kg.setter
    def kg(self, value):
        if value <= 0 or value > 50:
            raise ValueError(f"weight out of range: {value}")
        self._kg = value

    @property
    def items(self):
        return tuple(self._items)       # an immutable copy OUT
```

Listing 8.50 (Continued) *order_bug_hunt.py: the same hunt in Python, fixed with properties, a validating setter and tuples.*

```
class TestSafeOrder(unittest.TestCase):
    def test_caller_cannot_change_items(self):
        bag = ["shirt", "sarong"]
        order = SafeOrder("Sari", 3.5, bag)
        bag.append("stranger's jacket")
        self.assertEqual(("shirt", "sarong"), order.items)

    def test_bad_weight_refused(self):
        order = SafeOrder("Sari", 3.5, [])
        with self.assertRaises(ValueError):
            order.kg = -2
        self.assertEqual(3.5, order.kg)

    def test_customer_read_only_and_required(self):
        with self.assertRaises(AttributeError):
            SafeOrder("Sari", 1, []).customer = None
        with self.assertRaises(ValueError):
            SafeOrder(" ", 1, [])


if __name__ == "__main__":
    bag = ["shirt", "sarong"]
    leaky = LeakyOrder("Sari", 3.5, bag)
    leaky.customer = None
    bag.append("stranger's jacket")
    leaky.kg = -2
    leaky.items.clear()
    print(f"leaky: customer={leaky.customer}, kg={leaky.kg}, items={leaky.items}, caller's
bag={bag}")

    bag = ["shirt", "sarong"]
    safe = SafeOrder("Sari", 3.5, bag)
    bag.append("stranger's jacket")
    try:
        safe.kg = -2
    except ValueError as e:
        print("safe.kg = -2 refused:", e)
    print(f"safe : customer={safe.customer}, kg={safe.kg}, items={safe.items}")
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
leaky: customer=None, kg=-2, items=[], caller's bag=[]
safe.kg = -2 refused: weight out of range: -2
safe : customer=Sari, kg=3.5, items=('shirt', 'sarong')
test_bad_weight_refused (__main__.TestSafeOrder.test_bad_weight_refused) ... ok
test_caller_cannot_change_items (__main__.TestSafeOrder.test_caller_cannot_change_items)
... ok
test_customer_read_only_and_required
(__main__.TestSafeOrder.test_customer_read_only_and_required) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Line by line. The first output line shows all four bugs at once: no customer, a negative weight, and an empty item list, with the caller's `bag` empty too, because `leaky.getItems().clear()` cleared the very list that `bag` refers to. In `SafeOrder`, the constructor checks the customer, sends the weight through `setKg` so that the rule "between 0 and 50 kg" lives in one method only, and copies the incoming list; the later `bag.add("stranger's jacket")` changes only the caller's list. `getItems()` returns `Collections.unmodifiableList(items)`, a read-only *view*: reading works,

and any attempt to change it throws `UnsupportedOperationException`. Its return type is written `java.util.List<String>` in full because the chapter's folder also contains our own class named `List` (**Listing 8.3**); a simple name `List` in this file would mean ours. In Python (**Listing 8.50**), `kg` is a property with a validating setter, so even the assignment `self.kg = kg` inside `__init__` is checked; `customer` has no setter, so it is read-only; and `items` returns a *tuple*, which cannot be changed at all.

A tempting wrong solution. Making the list field `final` feels like a fix for bugs 2 and 4, and it is not. `final` means the *reference* can never be reassigned; the list it refers to can still be cleared, added to, and shared. `LeakyOrder.items` is already `final`, and the output shows it being emptied from outside. Immutability of an object requires copying and read-only views, not only `final` fields [2].

> **CHECK YOURSELF.** A class stores `private final int[] slots` received in its constructor and has `int[] getSlots() { return slots; }`. Name the two leaks and their fixes.
>
> *Answer.* The constructor aliases the caller's array (fix: `this.slots = slots.clone()` or `Arrays.copyOf(slots, slots.length)`), and the getter hands out the internal array (fix: return a copy the same way). Arrays have no read-only view, so copying is the only option.

8.9 The Problems of This Chapter Compared

Table 8.23 lays the twenty-one problems side by side. Read down the last column: every wrong solution fails on one particular kind of input, and in most problems that input is not the example given in the statement. That is the strongest argument this chapter can make for Step 7 of the method.

Table 8.23 The twenty-one problems of the chapter: what each practices and the single case its tempting wrong solution gets wrong.

Problem	Structure or topic	Main idea	The edge case the wrong solution misses
8.1	tree	build bottom-up	left and right child swapped
8.2	tree	change in place by delegation	missing subtree (`null`)
8.3	tree	pre-order traversal; testing printed output	order of node and subtrees
8.4	tree	deep copy	shared subtrees after `mirror()`
8.5	tree, array	return the last position used	right subtree start (off by one)
8.6	BST	one path, height-bounded	value absent (`null` subtree)
8.7	BST	attach the new leaf	leaf created but not attached
8.8	list	map	forgetting to recurse
8.9	list	pass a signal through	`-1` shifted into a valid position
8.10	list	logical AND, vacuous truth	negative element last
8.11	list	filter keeps order	negative element in the middle
8.12	list	compare neighbors	equal neighbors (`<` vs `<=`)
8.13	two lists	merge with two base cases	one list empty
8.14	list	filter by equality	`x` in the last position
8.15	list	reuse a helper; $\Theta(n^2)$	unsorted input; empty list
8.16	file, exceptions	`try` per line, file closed always	bad line in the middle; missing file
8.17	array, Javadoc	size vs capacity invariant	`index == size`
8.18	interface	default method, contract tests	rounding up; float truncation
8.19	hierarchy	overload at compile time, override at run time	declared type vs object's class
8.20	hierarchy	extract superclass, hook, `final`	sibling classes are not is-a
8.21	encapsulation	copy in, read-only out, validate	aliasing through `final` fields

8.10 Profitina in Practice

Profitina, an AI-visibility (GEO) platform in production at app.profitina.com, measures how ready a brand's website is to be cited by AI answer engines and then helps fix what it finds. Three ideas of this chapter are visible in the public shape of that product.

One interface, many implementations. The scanner examines a site through a set of *signals*, such as how easily its content can be extracted, whether it publishes structured data, and signs that it is technically trustworthy. Each signal is scored and explained independently, and the scanning code treats them all through one common interface, exactly as the loop in **Listing 8.43** treats every `PriceRule` alike. Adding a signal means adding a class, not editing the loop; testing a signal means running the same contract of tests against it, as `PriceRuleContract` does in **Listing 8.44**.

Untrusted input, parsed without crashing. A scanner reads pages written by strangers, and the community features screen every posted link with a rule floor that combines several signals. The situation is Problem 8.16 at a larger scale: most input is fine, some is malformed, and one bad item must cost one item, never the whole job. The discipline is the same: catch the specific failure

where it can be handled, record it with enough context to act on, and always release the resources that were opened.

Encapsulation as a legal duty, and tests as a release gate. Profitina's account features are built for Indonesia's personal data protection law, UU PDP 27/2022, with explicit consent, data export and full account deletion. Rules like these can only be enforced if personal data is changed in one controlled place, which is what `SafeOrder` in Problem 8.21 does for a laundry order. And no release goes out until the full automated test suite, well over a thousand tests, passes; the self-checking programs of this chapter are the same idea in miniature.

As always, the product advises and its users decide: a visibility score is a measurement with an explanation, not a guarantee. The runnable version of every idea above is the Profitina Laundry code of this chapter.

8.11 Common Mistakes, and How to Avoid Them

Forgetting the missing subtree. Every recursive tree method must handle `null` children: guard each delegation with `if (left != null)` (Python `is not None`), or make the base case the missing tree itself. Test with a chain and with a node that has only one child (Problems 8.2, 8.6).

Calling a method and ignoring its result. A recursive call whose value is thrown away (`left.createFreshCopy();`) does work for nothing. If a method returns something, the caller almost always needs it (Problems 8.4, 8.7).

A second base case that skips a check. "A one-element list is fine" is true only if you look at the element. Use the empty list as the only base case unless the combine step needs two elements (Problems 8.10, 8.14; the exception is 8.12).

Treating a signal value as data. `-1`, `null` and empty strings mean "no answer"; arithmetic on them produces believable nonsense. Test for the signal before using the value (Problem 8.9).

Off by one at a boundary. `<` versus `<=`, `>` versus `>=`, "last used" versus "next free". Write the convention in the documentation comment and put a test exactly on the boundary (Problems 8.5, 8.12, 8.17).

One `try` around everything. A `try` around a whole loop turns one bad record into a lost file, and `catch (Exception e) {}` hides every bug. Put the `try` around the smallest unit that can fail independently and catch the most specific exception (Problem 8.16).

Doing arithmetic on money in floating point. Convert to whole units first and use integer types; a cast from `double` truncates (Problem 8.18).

Confusing overloading and overriding. The overload is chosen by the compiler from the declared types; the override is chosen at run time from the object's class. Trace them in two separate columns, as in **Table 8.20**.

Forgetting `super` in an override. `return 2 * pricePerKg();` inside `pricePerKg` calls itself forever. Write `super.pricePerKg()` (Problem 8.19).

Inheritance for convenience. `extends` claims "is a". `NegateAll extends List` and `Enemy extends Player` are false claims that compile. Use a utility class or a common superclass instead (Sections 8.3.2 and 8.8.2).

Believing `final` makes an object immutable. `final` fixes the reference, not the object. Copy mutable inputs in, hand out read-only views or copies, and validate every change (Problem 8.21).

Testing only the example from the statement. In Problems 8.1, 8.10, 8.14, 8.15 and 8.17, a test built only from the ordinary example would have let the tempting wrong solution through. Write the edge-case tests first.

8.12 Chapter Summary

- A multi-class design problem is solved in eight steps (Section 8.2): collect nouns and verbs, choose classes, give every method an owner, choose the relationships, decide access levels, draw the design, write the tests with their edge cases, and implement one method at a time.
- A recursive method on a tree or a list is written with the delegation recipe: a base case, a delegation to the smaller parts, and the node's own job. For lists, Equation (8.4) is the template of every solution.
- Trees: build bottom-up (8.1); change in place by delegation (8.2); the position of the node's own job decides pre-, in- or post-order (8.3); a deep copy keeps the results of its recursive calls (8.4); array bookkeeping needs one written convention (8.5); a BST search and insertion follow one path of length at most the height (8.6, 8.7).
- Lists: map (8.8), search with a signal (8.9), logical AND with vacuous truth (8.10), filter (8.11, 8.14), neighbor comparison (8.12), merge with two base cases (8.13), and a reused helper at quadratic cost, Equation (8.6) (8.15).
- Exceptions and files: a checked exception per bad record, caught inside the loop; the file closed by try-with-resources or `with`; low-level errors translated into messages people can act on (8.16).
- Arrays and documentation: size and capacity, a class invariant, Javadoc and docstrings that state preconditions and exceptions, and a test exactly at `index == size` (8.17).
- Interfaces and tests: a default method written once; implementations that share validation; contract tests run against every implementation; JUnit 6 as the standard form of the same checks (8.18).
- Inheritance, polymorphism and access control: overloads resolved at compile time, overrides at run time (8.19); duplicated classes refactored into an abstract superclass with a final template method and a protected hook (8.20); encapsulation defended by private fields, validation, defensive copies and read-only views (8.21).

8.13 Exercises

Exercises 8.1 to 8.6 and 8.11 ask for code; their solutions are programs in Section 8.14. The others ask for reasoning and are answered in words.

1. **(Tree height.)** Write `int height()` for `Tree`, where a leaf has height 1. What is the height of the tree of **Figure 8.1**?
2. **(Tree sum.)** Write `int sum()` that adds all the nodes of a tree. What does it return for the tree of **Figure 8.1**?
3. **(Leaves.)** Write `int leaves()` that counts the leaves. What does it return for the tree of **Figure 8.1**?
4. **(BST minimum.)** Write `int min()` for a binary search tree, visiting as few nodes as possible. Which nodes does it visit in the tree of **Figure 8.6**?
5. **(List sum.)** Write `static int sum(List a)` using only the operations of `List`. What is the sum of `[2, -5, 8, 0, 8]`, and what should the sum of the empty list be?

6. **(Take.)** Write `static List take(int n, List a)`, the first `n` elements of `a`, or all of them if `a` is shorter. A negative `n` is an error. Which exception should it throw, checked or unchecked, and why?
7. **(Exceptions by hand.)** Predict, line by line, the output of **Listing 8.51** before running it. Then do the same for **Listing 8.52**, and explain the one line where the Python program needs extra code to behave like the Java one.
8. **(Dispatch by hand.)** In **Listing 8.45**, suppose `describe()` in `Service` were declared `final` and the override in `Express` removed. What would lines A (third pass), D and E print?
9. **(Encapsulation.)** A class `Receipt` has `private final ArrayList<Integer> amounts`, a constructor that stores the list it is given, a method `total()` that adds the amounts, and `public ArrayList<Integer> getAmounts()`. Describe a sequence of three statements, written outside the class, that makes `total()` return a different value without any method of `Receipt` being called in between, and fix the class.
10. **(Design.)** Profitina Laundry adds couriers. A courier has a name and a vehicle (motorcycle or car); a motorcycle carries at most 10 kg per trip and a car 40 kg. An order is assigned to one courier, and assigning an order that is too heavy for the courier's vehicle is an error. Apply Steps 1 to 6 of the method: list the classes, their fields and methods with access levels, the relationships, and the exception.
11. **(Persistent insertion.)** Write `Tree insertPersistent(int n)` for a BST that changes no existing node: it returns a new tree that shares every untouched subtree with the old one. How many new nodes does `insertPersistent(9)` create in the tree of **Figure 8.6**?
12. **(Checked or unchecked.)** In Problem 8.16, `OrderFormatException` is checked and `ListException` in Section 8.3.2 is unchecked. Give the rule that decides between the two, and say which kind you would choose for an exception thrown when a price rule receives a negative weight, and why.

The two programs of Exercise 7 are:

Listing 8.51 `TraceFinally.java`: a `try` with `catch` and `finally`, and an exception that the `catch` does not handle.

```java
// TraceFinally.java - Exercise 8.7: predict the output before you run it.
public class TraceFinally {

    static int parseKg(String text) {
        try {
            System.out.println("try " + text);
            int kg = Integer.parseInt(text);
            if (kg <= 0) {
                throw new IllegalArgumentException("not positive");
            }
            return kg;
        } catch (NumberFormatException e) {
            System.out.println("catch NumberFormatException");
            return -1;
        } finally {
            System.out.println("finally " + text);
        }
    }

    public static void main(String[] args) {
        System.out.println("result " + parseKg("4"));
        System.out.println("result " + parseKg("x"));
        try {
            System.out.println("result " + parseKg("0"));
        } catch (IllegalArgumentException e) {
            System.out.println("main caught " + e.getMessage());
        }
    }
}
```

Listing 8.52 `trace_finally.py`: the same flow in Python, where both errors are a `ValueError`.

```python
# trace_finally.py - Exercise 8.7: predict the output before you run it.


def parse_kg(text):
    try:
        print("try", text)
        kg = int(text)
        if kg <= 0:
            raise ValueError("not positive")
        return kg
    except ValueError as e:
        if str(e) == "not positive":
            raise                       # re-raise: not ours to handle
        print("catch ValueError")
        return -1
    finally:
        print("finally", text)


print("result", parse_kg("4"))
print("result", parse_kg("x"))
try:
    print("result", parse_kg("0"))
except ValueError as e:
    print("main caught", e)
```

8.14 Solutions to the Exercises

Exercises 1 to 6 and 11. **Listing 8.53** and **Listing 8.54** contain all the requested methods, each checked against the values worked out below.

Listing 8.53 `ExerciseSolutions.java`: height, sum, leaves, BST minimum, list sum, take and persistent insertion, with checks. Compile it together with `List.java` and `ListException.java`.

```
// ExerciseSolutions.java - solutions to the programming exercises of Chapter 8.
public class ExerciseSolutions {

    static final class Tree {
        private final int root;
        private final Tree left, right;

        Tree(int root, Tree left, Tree right) {
            this.root = root;
            this.left = left;
            this.right = right;
        }

        Tree(int root) {
            this(root, null, null);
        }

        /** Exercise 8.1: the number of levels; a leaf has height 1. */
        int height() {
            int l = (left == null) ? 0 : left.height();
            int r = (right == null) ? 0 : right.height();
            return 1 + Math.max(l, r);
        }

        /** Exercise 8.2: the sum of all nodes. */
        int sum() {
            return root + (left == null ? 0 : left.sum()) + (right == null ? 0 :
right.sum());
        }

        /** Exercise 8.3: the number of leaves. */
        int leaves() {
            if (left == null && right == null) {
                return 1;
            }
            return (left == null ? 0 : left.leaves()) + (right == null ? 0 :
right.leaves());
        }

        /** Exercise 8.4: the smallest value of a BST is at the end of the left spine. */
        int min() {
            return (left == null) ? root : left.min();
        }
```

Listing 8.53 (Continued) *ExerciseSolutions.java: height, sum, leaves, BST minimum, list sum, take and persistent insertion, with checks. Compile it together with List.java and ListException.java.*

```
        /** Exercise 8.11: insert without changing any existing node (path copying). */
        Tree insertPersistent(int n) {
            if (n < root) {
                return new Tree(root, left == null ? new Tree(n) :
left.insertPersistent(n), right);
            } else if (n > root) {
                return new Tree(root, left, right == null ? new Tree(n) :
right.insertPersistent(n));
            }
            return this;
        }

        @Override
        public String toString() {
            if (left == null && right == null) {
                return Integer.toString(root);
            }
            String l = (left == null) ? "-" : left.toString();
            String r = (right == null) ? "-" : right.toString();
            return root + "(" + l + "," + r + ")";
        }
    }

    /** Exercise 8.5: the sum of a List. */
    static int sum(List a) {
        return a.empty() ? 0 : a.head() + sum(a.tail());
    }

    /** Exercise 8.6: the first n elements (all of them if the list is shorter). */
    static List take(int n, List a) {
        if (n < 0) {
            throw new IllegalArgumentException("n must not be negative: " + n);
        }
        if (n == 0 || a.empty()) {
            return List.nil();
        }
        return List.cons(a.head(), take(n - 1, a.tail()));
    }

    private static int passed = 0;
```

Listing 8.53 (Continued) *ExerciseSolutions.java: height, sum, leaves, BST minimum, list sum, take and persistent insertion, with checks. Compile it together with List.java and ListException.java.*

```
    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        Tree t = new Tree(1,
                new Tree(2, new Tree(4, null, new Tree(8)), new Tree(5)),
                new Tree(3, new Tree(6), new Tree(7, new Tree(9), null)));
        check("8.1 height", 4, t.height());
        check("8.2 sum", 45, t.sum());
        check("8.3 leaves", 4, t.leaves());
        Tree bst = new Tree(5, new Tree(3, new Tree(1), new Tree(4)), new Tree(8, new
Tree(7), new Tree(10)));
        check("8.4 min", 1, bst.min());
        check("8.5 sum", 13, sum(List.of(2, -5, 8, 0, 8)));
        check("8.6 take 2", "[7, 5]", take(2, List.of(7, 5, 3)).toString());
        check("8.6 take 9", "[7, 5, 3]", take(9, List.of(7, 5, 3)).toString());
        Tree bigger = bst.insertPersistent(9);
        check("8.11 new tree", "5(3(1,4),8(7,10(9,-)))", bigger.toString());
        check("8.11 old tree kept", "5(3(1,4),8(7,10))", bst.toString());
        check("8.11 left half shared", true, bigger.left == bst.left);
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
PASS 8.1 height
PASS 8.2 sum
PASS 8.3 leaves
PASS 8.4 min
PASS 8.5 sum
PASS 8.6 take 2
PASS 8.6 take 9
PASS 8.11 new tree
PASS 8.11 old tree kept
PASS 8.11 left half shared
All 10 checks passed.
```

Listing 8.54 `exercise_solutions.py`: the same solutions in Python.

```
# exercise_solutions.py - solutions to the programming exercises of Chapter 8.
import unittest
from cons_list import List


class Tree:
    def __init__(self, root, left=None, right=None):
        self._root = root
        self._left = left
        self._right = right

    def height(self):
        """Exercise 8.1: the number of levels; a leaf has height 1."""
        l = 0 if self._left is None else self._left.height()
        r = 0 if self._right is None else self._right.height()
        return 1 + max(l, r)

    def sum(self):
        """Exercise 8.2: the sum of all nodes."""
        return (self._root
                + (0 if self._left is None else self._left.sum())
                + (0 if self._right is None else self._right.sum()))

    def leaves(self):
        """Exercise 8.3: the number of leaves."""
        if self._left is None and self._right is None:
            return 1
        return ((0 if self._left is None else self._left.leaves())
                + (0 if self._right is None else self._right.leaves()))

    def min(self):
        """Exercise 8.4: the smallest value of a BST ends the left spine."""
        return self._root if self._left is None else self._left.min()

    def insert_persistent(self, n):
        """Exercise 8.11: insert without changing any existing node."""
        if n < self._root:
            new_left = Tree(n) if self._left is None else self._left.insert_persistent(n)
            return Tree(self._root, new_left, self._right)
        if n > self._root:
            new_right = Tree(n) if self._right is None else
self._right.insert_persistent(n)
            return Tree(self._root, self._left, new_right)
        return self
```

Listing 8.54 (Continued) *exercise_solutions.py: the same solutions in Python.*

```
    def __str__(self):
        if self._left is None and self._right is None:
            return str(self._root)
        l = "-" if self._left is None else str(self._left)
        r = "-" if self._right is None else str(self._right)
        return f"{self._root}({l},{r})"


def list_sum(a):
    """Exercise 8.5: the sum of a List."""
    return 0 if a.empty() else a.head() + list_sum(a.tail())


def take(n, a):
    """Exercise 8.6: the first n elements (all of them if the list is shorter)."""
    if n < 0:
        raise ValueError(f"n must not be negative: {n}")
    if n == 0 or a.empty():
        return List.nil()
    return List.cons(a.head(), take(n - 1, a.tail()))


class TestExercises(unittest.TestCase):
    def setUp(self):
        self.t = Tree(1, Tree(2, Tree(4, None, Tree(8)), Tree(5)),
                      Tree(3, Tree(6), Tree(7, Tree(9))))
        self.bst = Tree(5, Tree(3, Tree(1), Tree(4)), Tree(8, Tree(7), Tree(10)))

    def test_tree_measures(self):
        self.assertEqual((4, 45, 4), (self.t.height(), self.t.sum(), self.t.leaves()))
        self.assertEqual(1, self.bst.min())

    def test_list_functions(self):
        self.assertEqual(13, list_sum(List.of(2, -5, 8, 0, 8)))
        self.assertEqual("[7, 5]", str(take(2, List.of(7, 5, 3))))
        self.assertEqual("[7, 5, 3]", str(take(9, List.of(7, 5, 3))))

    def test_persistent_insert(self):
        bigger = self.bst.insert_persistent(9)
        self.assertEqual("5(3(1,4),8(7,10(9,-)))", str(bigger))
        self.assertEqual("5(3(1,4),8(7,10))", str(self.bst))
        self.assertIs(self.bst._left, bigger._left)
```

Listing 8.54 (Continued) *exercise_solutions.py: the same solutions in Python.*

```
if __name__ == "__main__":
    unittest.main(verbosity=2)
```

```
test_list_functions (__main__.TestExercises.test_list_functions) ... ok
test_persistent_insert (__main__.TestExercises.test_persistent_insert) ... ok
test_tree_measures (__main__.TestExercises.test_tree_measures) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

1. `height()` is 1 plus the larger height of the two subtrees, with 0 for a missing subtree. The longest path in **Figure 8.1** is 1, 2, 4, 8 (or 1, 3, 7, 9), so the height is 4.
2. The nodes are 1 to 9, so the sum is $\frac{9 \cdot 10}{2} = 45$.
3. The leaves are 5, 6, 8 and 9: four. A node is a leaf exactly when both subtrees are missing.

4. The smallest value of a BST is at the end of the path that always goes left: 5, 3, 1. Three visits; the right subtrees are never touched.
5. $2 - 5 + 8 + 0 + 8 = 13$. The sum of the empty list is 0, the neutral element of addition, just as the empty list was `true` for the logical AND of Problem 8.10.
6. See `take` in the listings. A negative `n` is a programming error by the caller, not a situation a correct program must be ready for, so an *unchecked* `IllegalArgumentException` (Python `ValueError`) is right.
7. The real output of **Listing 8.51** is below; **Listing 8.52** prints the same lines except `catch ValueError` in the fifth line.

```
try 4
finally 4
result 4
try x
catch NumberFormatException
finally x
result -1
try 0
finally 0
main caught not positive
```

For `"4"`: `try`, then `return 4` is *prepared*, then `finally` runs before the method really returns, then `main` prints `result 4`. For `"x"`: `parseInt` throws, the `catch` prints and prepares `return -1`, `finally` runs, `main` prints `result -1`. For `"0"`: the `IllegalArgumentException` is not a `NumberFormatException`, so the `catch` does not match; `finally` still runs, and the exception travels up to `main`, where it is caught. In Python both errors are `ValueError`, so the `except` clause catches both; the extra lines `if str(e) == "not positive": raise` re-raise the second one so that it reaches the caller, as in Java. A cleaner Python design would define its own exception class for "not positive".

8. A `final` method cannot be overridden, so `describe()` is always the `Service` version, which prints `code @ pricePerKg/kg`. `pricePerKg()` is still overridden. Line A (third pass): `EXPRESS @ 14000/kg, 3 kg = 42000`. Line D: `service: EXPRESS @ 14000/kg`. Line E: `washing: EXPRESS @ 14000/kg`. The overloads are chosen exactly as before; only the "(same day)" disappears.
9. `ArrayList<Integer> list = new ArrayList<>(); Receipt r = new Receipt(list); list.add(1000000);` now `r.total()` includes a million rupiah nobody paid. (Equally, `r.getAmounts().add(...)`.) Fix: copy in the constructor (`this.amounts = new ArrayList<>(amounts)`) and return `Collections.unmodifiableList(amounts)`, declared as `java.util.List<Integer>`.
10. Classes: `Courier` (`- name : String`, `- vehicle : Vehicle`, `+ assign(order : Order)`, `+ getAssigned()`), `Vehicle` as an interface or abstract class with `+ maxKg() : double`, implemented by `Motorcycle` (10) and `Car` (40), and the existing `Order`. Relationships: courier *has-a* vehicle; courier *has* a list of assigned orders (copied out read-only, as in Problem 8.21); motorcycle and car *can-do* `Vehicle`. `assign` checks `order.getKg() <= vehicle.maxKg()` and otherwise throws an `OverweightException`; checked, if assigning is something callers must plan for (a dispatcher choosing another courier), unchecked if a correct caller always checks first.
11. See `insertPersistent`. Inserting 9 into the tree of **Figure 8.6** creates four new nodes: new copies of 5, 8 and 10 along the path, and the leaf 9. The left half `3(1,4)` and the leaf 7 are shared, as the check `8.11 left half shared` confirms, and the old tree still prints `5(3(1,4),8(7,10))`. Sharing is safe only because no one changes nodes after creation, which is why this design suits the immutable `List` of Section 8.3.2 as well.

12. The rule [2]: use a *checked* exception for conditions a correct program must expect and can recover from (a malformed line in a file typed by a person, a missing file); use an *unchecked* exception for a broken precondition, a programming error in the caller (`head()` of the empty list, a negative index). A negative weight passed to `priceFor` is a caller's error, because weights should have been validated when the order was read, so `IllegalArgumentException`, unchecked, as in **Listing 8.43**.

References

[1] Python Software Foundation, "unittest — Unit testing framework," *The Python Standard Library*, Python 3.14. [Online]. Available: https://docs.python.org/3/library/unittest.html

[2] J. Bloch, *Effective Java*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2018.

[3] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2005.

[4] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, MA, USA: MIT Press, 1996.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[6] Oracle, "The Java Tutorials: Exceptions" and "The try-with-resources Statement." [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions/

[7] Python Software Foundation, "Errors and Exceptions" and "Reading and Writing Files," *The Python Tutorial*, Python 3.14. [Online]. Available: https://docs.python.org/3/tutorial/

[8] Python Software Foundation, "abc — Abstract Base Classes" and "functools — Higher-order functions and operations on callable objects," *The Python Standard Library*, Python 3.14. [Online]. Available: https://docs.python.org/3/library/

[9] The JUnit Team, *JUnit User Guide*, version 6.1.3, Aug. 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://docs.junit.org/current/

[10] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*. Oracle America, 2023. [Online]. Available: https://docs.oracle.com/javase/specs/

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

[12] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2018.

The rule for choosing checked and unchecked exceptions, the advice on defensive copies and immutability, and the preference for interfaces over inheritance follow Bloch [2]; the resolution of overloaded methods at compile time and the four access levels are defined in the language specification [10], and exception handling with try-with-resources is described in the Java tutorial [6]. The Python tests use `unittest` [1], abstract base classes and `singledispatch` [8], and file handling with `with` [7]; the JUnit 6 versions follow the user guide [9]. The extract-superclass refactoring of Problem 8.20 is from Fowler [12], and the template method of Problems 8.19 and 8.20 from Gamma et al. [11]. Cons lists and the link between recursion and induction come from Abelson and Sussman [4]; binary search trees, merging and the growth rates of Problems 8.6, 8.13 and 8.15 are treated in Cormen et al. [5]; and the class diagrams use the notation of the UML user guide [3].

Chapter 9
Collections and Generics

By the end of this chapter you will be able to choose, for any job that involves many objects, the container that does that job well, and defend the choice with numbers. You will use Java's Collections Framework — the interfaces `List`, `Set`, `Map`, `Queue` and `Deque` and their workhorse classes `ArrayList`, `LinkedList`, `HashSet`, `LinkedHashSet`, `TreeSet`, `HashMap`, `LinkedHashMap`, `TreeMap`, `ArrayDeque` and `PriorityQueue` — side by side with Python's `list`, `tuple`, `set`, `frozenset`, `dict`, `deque` and `heapq`. You will walk through a collection safely, remove from it without tripping over your own feet, and sort it by any rule with `Comparable`, `Comparator` and Python's `key=` functions. You will learn generics from zero: what a type parameter is, how to write a generic class and a generic method, what a bounded type such as `<T extends Comparable<T>>` buys you, why a `List<Cat>` is not a `List<Animal>` and how the wildcards `? extends` and `? super` repair that, and what *type erasure* leaves behind when the program runs; and you will meet the same ideas in Python's `typing` module, checked by the tool mypy. Along the way we pick up two small tools that the rest of the chapter needs: constants declared in an interface, and recursive methods that use a helper. Everything comes together in an order book for Profitina Laundry that finds any order by its number, by its customer or by its status, keeps a queue of pending washes with express orders first, and prints sorted revenue reports.

9.1 Where We Left Off

Chapter 5 gave us arrays and our first growing list, `ArrayList`, and warned that the mysterious `<String>` in `ArrayList<String>` would be explained in Chapter 9. This is that chapter. Chapter 6 showed that an *interface* is a promise without a body, that `List` is such an interface and `ArrayList` one of its implementations, and that a class can promise a natural order by implementing `Comparable`. Chapter 7 built class hierarchies, so that one variable of type `Service` could hold a wash, an iron or an express service, and Chapter 8 practised all of this on recursive structures, including a small immutable list of integers built from `nil` and `cons`.

Now look at the laundry at the end of a busy Saturday. There are a few hundred orders. The owner asks four questions in one breath: *Where is order L-217? What has Bu Sari left with us? Which orders are ready to be collected? Which bags go into the next free machine?* With what we know so far, each question is a loop over one big list of orders, checking every order in turn. That works for ten orders. For ten thousand orders, asked a few hundred times a day, it is the difference between an answer that appears at once and a counter queue that reaches the street.

The cure is not a cleverer loop but a better *container*: a *map* from order numbers to orders for "find L-217", a map from customers to their orders for "what has Bu Sari left", a *set* per status for "which are ready", and a *priority queue* for "which bag next, express first". Java calls this family of containers the **Collections Framework**; Python builds most of it into the language. This chapter studies the family member by member, then shows how to pick the right member by counting the work each one does.

The second half of the story is how one container class can hold `String`s in one program and `LaundryOrder`s in another, with the compiler still checking every use. That is **generics**, the idea behind the `<String>`.

> ✓ **KEY INSIGHT.** A collection is an answer to a *question you will ask often*. Ask "what will I look things up by, and in what order do I need them?" before you ask "which class should I write `new` in front of?". The question chooses the collection; the collection makes the question cheap.

9.2 Two Tools to Carry Along: Constants in Interfaces and Recursive Helpers

Before we open the collections we pick up two small tools. The first is a fact about interfaces that Chapter 6 did not need: an interface can hold *constants* as well as method headers. The second is a way of writing recursive methods that keeps their callers simple, which we will use for a generic list later in the chapter.

9.2.1 Constants in interfaces

Some values never change while a program runs: the number of kilograms in a pound, the points awarded for a win, the codes that stand for the outcome of a match. Chapter 4 wrote such values as `public static final` fields of a class, for example `KILOGRAMS_PER_POUND`. An interface may hold them too. Inside the interface `Measurable` we can write

```
int LOSE = 0;
int DRAW = 1;
int WIN = 2;
```

and use them anywhere as `Measurable.WIN`. You must not declare these fields `public` or `private` yourself: every field of an interface is **automatically** `public static final` [1]. `public`, because an interface is a public promise; `static`, because an interface has no objects and so the value belongs to the interface itself; `final`, because an interface cannot hold changing state. A class that implements the interface inherits the constants and may use them without the prefix, just as it inherits the method headers.

Listing 9.1 puts the idea to work. `Measurable` carries the three outcome codes and the method `getMeasure()` from Chapter 6; `Team` implements it and measures a team by its league points; a second interface, `Units`, holds a conversion factor.

Listing 9.1 `InterfaceConstants.java`: constants in the interfaces `Measurable` and `Units`, used with and without a prefix, and an `enum` as the modern alternative.

```
import java.util.Locale;

public class InterfaceConstants {
    public static void main(String[] args) {
        System.out.println("LOSE = " + Measurable.LOSE + ", DRAW = " + Measurable.DRAW
                + ", WIN = " + Measurable.WIN);

        int[] results = {Measurable.WIN, Measurable.DRAW, Measurable.WIN, Measurable.LOSE};
        Team team = new Team("Surabaya Stars", results);
        System.out.println(team.getName() + " scored " + team.getMeasure() + " points");

        double pounds = 11.0;
        System.out.println(String.format(Locale.US, "%.1f lb = %.3f kg",
                pounds, pounds * Units.KG_PER_POUND));

        // Measurable.WIN = 5;   // does not compile: WIN is final

        // An enum is the modern home for a fixed set of named values.
        for (Outcome o : Outcome.values()) {
            System.out.println(o + ": " + o.points() + " pt");
        }
        System.out.println("Outcome.valueOf(\"DRAW\") = " + Outcome.valueOf("DRAW"));
    }
}

interface Measurable {
    int LOSE = 0;          // automatically public static final
    int DRAW = 1;
    int WIN = 2;

    double getMeasure();
}

interface Units {
    double KG_PER_POUND = 0.45359237;   // exact, by international agreement
}

class Team implements Measurable {
    private final String name;
    private final int[] results;

    Team(String name, int[] results) {
        this.name = name;
        this.results = results.clone();
    }
```

Listing 9.1 (Continued) *InterfaceConstants.java: constants in the interfaces Measurable and Units, used with and without a prefix, and an enum as the modern alternative.*

```
    String getName() {
        return name;
    }

    @Override
    public double getMeasure() {
        int points = 0;
        for (int r : results) {
            if (r == WIN) {             // inherited constant: no prefix needed
                points += 3;
            } else if (r == DRAW) {
                points += 1;
            }
        }
        return points;
    }
}

enum Outcome {
    LOSE(0), DRAW(1), WIN(3);

    private final int points;

    Outcome(int points) {
        this.points = points;
    }

    int points() {
        return points;
    }
}
```

```
LOSE = 0, DRAW = 1, WIN = 2
Surabaya Stars scored 7.0 points
11.0 lb = 4.990 kg
LOSE: 0 pt
DRAW: 1 pt
WIN: 3 pt
Outcome.valueOf("DRAW") = DRAW
```

Read the listing from the top. In `main`, `Measurable.LOSE`, `Measurable.DRAW` and `Measurable.WIN` are read through the interface name, exactly like static fields of a class. The array `results` holds four match outcomes, and `Team.getMeasure()` turns them into league points: three for a win, one for a draw. Inside `Team` the comparisons say `r == WIN` and `r == DRAW` without the prefix, because `Team implements Measurable` and so inherits the names. Two wins and a draw give $3 + 1 + 3 + 0 = 7$ points. The conversion uses `Units.KG_PER_POUND`, the exact international value 0.45359237, so eleven pounds are $11 \times 0.45359237 = 4.98951607$ kg, printed to three places as `4.990`.

The commented line `// Measurable.WIN = 5;` shows what `final` means. Remove the two slashes and `javac` refuses the program:

```
InterfaceConstants.java:16: error: cannot assign a value to static final variable WIN
        Measurable.WIN = 5;
                   ^
1 error
```

Interface constants have one weakness: the outcome codes are plain `int`s, so nothing stops a caller from passing `7`, and nothing ties a win's points to the code for a win. Java's **enum** (enumeration) solves both: `enum Outcome { LOSE(0), DRAW(1), WIN(3); ... }` declares a type

with exactly three possible values, each an object that can carry data (`points`) and methods (`points()`). A method that takes an `Outcome` can receive nothing else. `Outcome.values()` lists the constants in declaration order, and `Outcome.valueOf("DRAW")` finds one by name. The advice of the standard reference on Java style is plain: use interfaces to define types, and put a group of related constants in an enum or a small utility class, not in an interface that classes implement merely to inherit the names [2]. Interface constants remain worth knowing, because much existing code uses them.

Python has no `final` keyword and no interface constants in Java's sense. **Listing 9.2** shows the three things it has instead.

Listing 9.2 `interface_constants.py`: class attributes marked `Final`, a module constant, and an `Enum`, whose members cannot be reassigned.

```
"""interface_constants.py: named constants in Python, and why an Enum is safer."""
from abc import ABC, abstractmethod
from enum import Enum
from typing import Final

KG_PER_POUND: Final = 0.45359237   # exact, by international agreement


class Measurable(ABC):
    LOSE: Final = 0   # Final is a promise checked by mypy, not by Python
    DRAW: Final = 1
    WIN: Final = 2

    @abstractmethod
    def get_measure(self) -> float: ...


class Team(Measurable):
    def __init__(self, name, results):
        self.name = name
        self._results = list(results)

    def get_measure(self):
        points = 0
        for r in self._results:
            if r == self.WIN:          # inherited class attribute
                points += 3
            elif r == self.DRAW:
                points += 1
        return float(points)


class Outcome(Enum):
    LOSE = 0
    DRAW = 1
    WIN = 3

    def points(self):
        return self.value


def main() -> None:
    print(f"LOSE = {Measurable.LOSE}, DRAW = {Measurable.DRAW}, WIN = {Measurable.WIN}")
```

Listing 9.2 (Continued) *interface_constants.py: class attributes marked Final, a module constant, and an Enum, whose members cannot be reassigned.*

```
    results = [Measurable.WIN, Measurable.DRAW, Measurable.WIN, Measurable.LOSE]
    team = Team("Surabaya Stars", results)
    print(f"{team.name} scored {team.get_measure()} points")

    pounds = 11.0
    print(f"{pounds:.1f} lb = {pounds * KG_PER_POUND:.3f} kg")

    # An enum is the modern home for a fixed set of named values.
    for o in Outcome:
        print(f"{o.name}: {o.points()} pt")
    print(f'Outcome["DRAW"] = {Outcome["DRAW"].name}')

    # Python does not stop this assignment at run time ...
    Measurable.WIN = 5
    print("after Measurable.WIN = 5:", team.get_measure(), "points")
    # ... but an Enum member cannot be reassigned.
    try:
        Outcome.WIN = 5
    except AttributeError as e:
        print("AttributeError:", e)


if __name__ == "__main__":
    main()
```

```
LOSE = 0, DRAW = 1, WIN = 2
Surabaya Stars scored 7.0 points
11.0 lb = 4.990 kg
LOSE: 0 pt
DRAW: 1 pt
WIN: 3 pt
Outcome["DRAW"] = DRAW
after Measurable.WIN = 5: 1.0 points
AttributeError: cannot reassign member 'WIN'
```

The first seven lines match Java exactly. The last two show the difference. `Measurable.WIN = 5` runs without complaint: a class attribute in Python is an ordinary variable that anyone may rebind. Worse, it silently breaks the team's score, which drops from 7 to 1, because the list `results` still holds the old code 2 while `WIN` now means 5, so only the draw is recognized. The annotation `Final` is a promise that the *type checker* enforces, not the interpreter. Run mypy, the standard static checker for Python type hints [3], and it reports both assignments before the program ever runs:

```
interface_constants.py:58: error: Cannot assign to final attribute "WIN"  [misc]
interface_constants.py:62: error: Cannot assign to final attribute "WIN"  [misc]
Found 2 errors in 1 file (checked 1 source file)
```

The `Enum` member, by contrast, protects itself at run time: `Outcome.WIN = 5` raises `AttributeError`. As in Java, an enum is the safest home for a fixed set of named values.

> ⚠ **WARNING.** mypy checks only functions that carry type hints. That is why `main` in **Listing 9.2** is declared `def main() -> None:`; without the `-> None`, mypy would skip its body and report nothing. When you want a Python program checked, annotate the functions you want checked.

9.2.2 Recursion, once more: base case and recursive call

Chapter 8 used recursion on trees and lists. Here we look at it once more on plain numbers, because one of its techniques, the *helper method*, reappears in the generic list of Section 9.8.

A recursive method is simply a method that calls itself. It can give a far more elegant solution than a loop, and it can also be baffling at first sight: it is easy to stare at a recursive method and lose track of what is happening, while the loop version reads straight down the page. The best way to think about recursion is that it splits one large problem into a smaller problem of the same kind plus a little work — *divide and conquer*. Every recursive method has two parts:

- the **base case**, the simplest version of the problem, answered directly; it is the deepest layer of the calls and it is what stops the repetition;
- the **recursive call**, in which the method calls itself with a *different*, smaller argument, so that the process keeps moving towards the base case.

Leave out the base case, or write a recursive call that does not move towards it, and the calls never stop: Java throws `StackOverflowError`, Python `RecursionError`.

The smallest example counts down. `printNums(x)` prints `x,` and then asks `printNums(x - 1)` to print the rest; `printNums(0)` is the base case and prints just `0`. So `printNums(10)` prints `10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0`.

The classic example is the Fibonacci sequence 0,1,1,2,3,5,8,13, …, in which each number is the sum of the two before it:

$$F(0) = 0, \qquad F(1) = 1, \qquad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2 \tag{9.1}$$

Equation (9.1) can be typed into Java almost word for word, and that "standard" version appears in **Listing 9.3** as `fibonacci(int n)`: two base cases, then two recursive calls. It is correct. It is also astonishingly slow, and the reason is worth seeing. **Figure 9.1** draws every call made by `fibonacci(5)`.

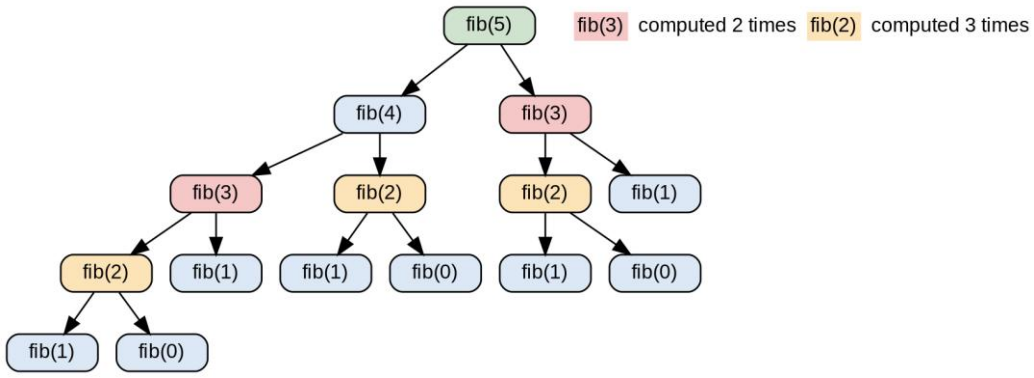


Figure 9.1 The calls made by the standard recursive `fibonacci(5)`: fifteen calls, because the same subproblems are solved again and again — `fib(3)` twice and `fib(2)` three times.

`fib(5)` asks for `fib(4)` and `fib(3)`; but `fib(4)` *also* asks for `fib(3)`, and nobody remembers the first answer. Let $C(n)$ be the number of calls made to compute $F(n)$. One call is made for n itself, plus the calls of its two recursive calls:

$$C(0) = C(1) = 1, \qquad C(n) = 1 + C(n-1) + C(n-2) \quad \Rightarrow \quad C(n) = 2F(n+1) - 1 \tag{9.2}$$

The closed form on the right is easy to check by induction: it holds for $n = 0$ and $n = 1$ because $2F(1) - 1 = 2F(2) - 1 = 1$, and if it holds for $n-1$ and $n-2$ then $C(n) = 1 + (2F(n) - 1) + (2F(n-1) - 1) = 2(F(n) + F(n-1)) - 1 = 2F(n+1) - 1$. For $n = 5$ it gives $2 \times 8 - 1 = 15$, the fifteen boxes of **Figure 9.1**. For $n = 30$ it gives $2 \times 1{,}346{,}269 - 1 = 2{,}692{,}537$ calls to

compute a number that a child can compute with thirty additions. Since $F(n)$ grows like $\varphi^n/\sqrt{5}$ with $\varphi = (1+\sqrt{5})/2 \approx 1.618$, the work grows exponentially: every extra step multiplies it by about 1.6.

9.2.3 Helper methods with extra parameters

The slow version recomputes because each call forgets what the others learned. The fix is to *carry* the two most recent values along as parameters, so that each step needs only one recursive call:

```
static long fibonacci(int n, long sum, long prev, int count) {
    if (count >= n - 1) {
        return sum + prev;
    }
    return fibonacci(n, sum + prev, sum, count + 1);  // one call, not two
}
```

`sum` holds $F(count)$ and `prev` holds $F(count-1)$. Each call moves one step along the sequence — the new `sum` is `sum + prev`, the new `prev` is the old `sum` — until `count` reaches $n-1$, when `sum + prev` is $F(n)$. The recursive call is the very last thing the method does, which is called **tail recursion**. The whole computation makes about n calls instead of $2F(n+1)-1$.

Its weakness is the call. To compute $F(10)$ we must write `fibonacci(10, 1, 0, 1)`, and a caller who types `fibonacci(10, 0, 0, 0)` gets a wrong answer without any warning. The solution is a **helper method**: a second, simple method that supplies the extra parameters and is the one callers actually use. In **Listing 9.3** it is called `fib(int n)`; it returns `fibonacci(n, 1, 0, 1)`, and the caller writes just `fib(10)`. The same idea appears in any recursive method that needs an *accumulator*: a public method with a friendly signature, and a private helper that does the recursion with the extra state. The `toString` of the generic list in Section 9.8.2 is built exactly this way.

A careful reader will notice that the base case is written `count >= n - 1`, not `count == n - 1`. With `==`, the call for $n = 1$ starts with `count` equal to 1 and $n - 1 = 0$; `count` only grows, so it never equals 0, the base case is never reached, and the program dies with `StackOverflowError`. The `>=` makes the base case reachable for every $n \geq 1$, and `fib` handles $n = 0$ itself. *A base case is only a base case if every chain of calls reaches it.*

A third way returns *two* values from each call, the pair $\{F(n), F(n-1)\}$; the pair for n is one addition away from the pair for $n-1$. This linear-time algorithm is a favorite of Goodrich, Tamassia and Goldwasser [4]; in Java it returns a `long[]` of length 2.

Listing 9.3 `Fibonacci.java`: a countdown, the standard two-call Fibonacci with a call counter, a tail-recursive version behind a helper method, and a linear version that returns two values at once.

```java
public class Fibonacci {
    static long calls = 0;   // how many times fibonacci(int) has been entered

    // Counts down to 0: base case, then a recursive call on a smaller problem.
    static void printNums(int x) {
        if (x == 0) {
            System.out.println(0);
        } else {
            System.out.print(x + ", ");
            printNums(x - 1);
        }
    }

    // The "standard" version: two recursive calls per step.
    static long fibonacci(int n) {
        calls++;
        if (n == 0) {
            return 0;
        } else if (n == 1) {
            return 1;
        } else {
            return fibonacci(n - 1) + fibonacci(n - 2);
        }
    }

    // Tail-recursive version: the extra parameters carry the work along.
    static long fibonacci(int n, long sum, long prev, int count) {
        if (count >= n - 1) {
            return sum + prev;
        }
        return fibonacci(n, sum + prev, sum, count + 1);
    }

    // The helper method hides the extra parameters from the caller.
    static long fib(int n) {
        if (n == 0) {
            return 0;
        }
        return fibonacci(n, 1, 0, 1);
    }
```

Listing 9.3 (Continued) *Fibonacci.java: a countdown, the standard two-call Fibonacci with a call counter, a tail-recursive version behind a helper method, and a linear version that returns two values at once.*

```
    // Linear time by returning two values at once: {F(n), F(n-1)}.
    static long[] fibonacciPair(int n) {
        if (n <= 1) {
            long[] answer = {n, 0};
            return answer;
        } else {
            long[] tmp = fibonacciPair(n - 1);
            long[] answer = {tmp[0] + tmp[1], tmp[0]};
            return answer;
        }
    }

    public static void main(String[] args) {
        printNums(10);

        for (int n : new int[] {10, 20, 30}) {
            calls = 0;
            long f = fibonacci(n);
            System.out.println("fibonacci(" + n + ") = " + f + " after " + calls + "
calls");
        }
        System.out.println("fibonacci(10, 1, 0, 1) = " + fibonacci(10, 1, 0, 1));
        System.out.println("fib(10) = " + fib(10) + ", fib(1) = " + fib(1) + ", fib(0) = "
+ fib(0));
        System.out.println("fibonacciPair(10)[0] = " + fibonacciPair(10)[0]);
        System.out.println("fibonacciPair(90)[0] = " + fibonacciPair(90)[0]);
    }
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
fibonacci(10) = 55 after 177 calls
fibonacci(20) = 6765 after 21891 calls
fibonacci(30) = 832040 after 2692537 calls
fibonacci(10, 1, 0, 1) = 55
fib(10) = 55, fib(1) = 1, fib(0) = 0
fibonacciPair(10)[0] = 55
fibonacciPair(90)[0] = 2880067194370816120
```

The call counts in the output are exactly those of Equation (9.2): $2F(11) - 1 = 2 \times 89 - 1 = 177$, $2F(21) - 1 = 21{,}891$ and $2F(31) - 1 = 2{,}692{,}537$. The other three methods agree with the slow one and compute $F(90) = 2{,}880{,}067{,}194{,}370{,}816{,}120$ in ninety steps; the slow method would need about 9×10^{18} calls, several hundred years on a laptop.

Notice the awkward part of `fibonacciPair`: it returns an array just because a method can return only one thing, and the caller must remember that index 0 means $F(n)$ and index 1 means $F(n-1)$. A small class that holds two values of chosen types, a `Pair`, would say what it means. Section 9.8.3 writes that class, once, for *any* two types — which is precisely the problem generics solve.

Listing 9.4 is the Python version.

Listing 9.4 `fibonacci.py`: the same four functions in Python, plus the standard version made fast by `functools.lru_cache`.

```python
"""fibonacci.py: recursion, helper functions, and a linear-time Fibonacci."""
from functools import lru_cache

calls = 0   # how many times fibonacci() has been entered


def print_nums(x):
    """Counts down to 0: base case, then a recursive call on a smaller problem."""
    if x == 0:
        print(0)
    else:
        print(x, end=", ")
        print_nums(x - 1)


def fibonacci(n):
    """The "standard" version: two recursive calls per step."""
    global calls
    calls += 1
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)


def fibonacci_tail(n, total, prev, count):
    """Tail-recursive version: the extra parameters carry the work along."""
    if count >= n - 1:
        return total + prev
    return fibonacci_tail(n, total + prev, total, count + 1)


def fib(n):
    """The helper function hides the extra parameters from the caller."""
    if n == 0:
        return 0
    return fibonacci_tail(n, 1, 0, 1)
```

Listing 9.4 (Continued) *fibonacci.py: the same four functions in Python, plus the standard version made fast by functools.lru_cache.*

```
def fibonacci_pair(n):
    """Linear time by returning two values at once: (F(n), F(n-1))."""
    if n <= 1:
        return (n, 0)
    f1, f2 = fibonacci_pair(n - 1)
    return (f1 + f2, f1)


@lru_cache(maxsize=None)
def fibonacci_cached(n):
    """The standard version again, but every answer is remembered."""
    return n if n <= 1 else fibonacci_cached(n - 1) + fibonacci_cached(n - 2)


def main():
    global calls
    print_nums(10)

    for n in (10, 20, 30):
        calls = 0
        f = fibonacci(n)
        print(f"fibonacci({n}) = {f} after {calls} calls")
    print("fibonacci(10, 1, 0, 1) =", fibonacci_tail(10, 1, 0, 1))
    print(f"fib(10) = {fib(10)}, fib(1) = {fib(1)}, fib(0) = {fib(0)}")
    print("fibonacciPair(10)[0] =", fibonacci_pair(10)[0])
    print("fibonacciPair(90)[0] =", fibonacci_pair(90)[0])
    print("fibonacci_cached(90) =", fibonacci_cached(90))


if __name__ == "__main__":
    main()
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
fibonacci(10) = 55 after 177 calls
fibonacci(20) = 6765 after 21891 calls
fibonacci(30) = 832040 after 2692537 calls
fibonacci(10, 1, 0, 1) = 55
fib(10) = 55, fib(1) = 1, fib(0) = 0
fibonacciPair(10)[0] = 55
fibonacciPair(90)[0] = 2880067194370816120
fibonacci_cached(90) = 2880067194370816120
```

Python has no overloading (Chapter 6), so the tail-recursive function needs its own name, `fibonacci_tail`; the printed labels were kept identical to Java's so that the outputs can be compared. Python returns two values naturally as a *tuple*, `(n, 0)`, unpacked with `f1, f2 = fibonacci_pair(n - 1)`. And the decorator `@lru_cache` makes a function *remember* every answer in a dictionary keyed by the arguments, so the naive two-call definition runs in linear time — a collection answering "have I seen this before?" in one step.

> ⚠ **WARNING.** Neither Java nor Python eliminates tail calls: every recursive call occupies a stack frame until it returns, and Python stops at a depth of about 1000 by default. Use recursion where the depth is small, as in trees; for a million steps, write a loop.

9.3 The Collections Framework: a Map of the Territory

A **collection** is an object that holds other objects, its *elements*, and offers operations on them as a group: add one, remove one, ask whether one is present, count them, visit them all. An array is a collection of the most primitive kind. Java's **Collections Framework**, in the package `java.util`, is

a set of interfaces that name the useful *kinds* of collection, plus a set of classes that implement those interfaces in different ways, plus two utility classes full of static helpers [5].

Figure 9.2 is the map of the territory. Read it from the top. The interface `Collection` promises what every group of elements can do: `add`, `remove`, `contains`, `size`, `isEmpty`, `iterator`. Three interfaces extend it: `List` (a sequence with positions), `Queue` (elements waiting to be taken out in some order) and `Set` (no duplicates). `Deque` extends `Queue` with work at both ends, and `SortedSet` extends `Set` with order. Below them, joined by dashed arrows meaning *implements*, are the concrete classes you create with `new`; a solid arrow between two classes means *extends*, as `LinkedHashSet` extends `HashSet`. Below the collections, `Map` stands apart: a map is not a collection of elements but a collection of *key* → *value* associations, so it has its own root interface, with `SortedMap` below it. Bottom right are the two utility classes `Arrays` and `Collections`, which extend nothing but `Object` and consist entirely of static methods such as `Arrays.asList`, `Collections.sort` and `Collections.max`.

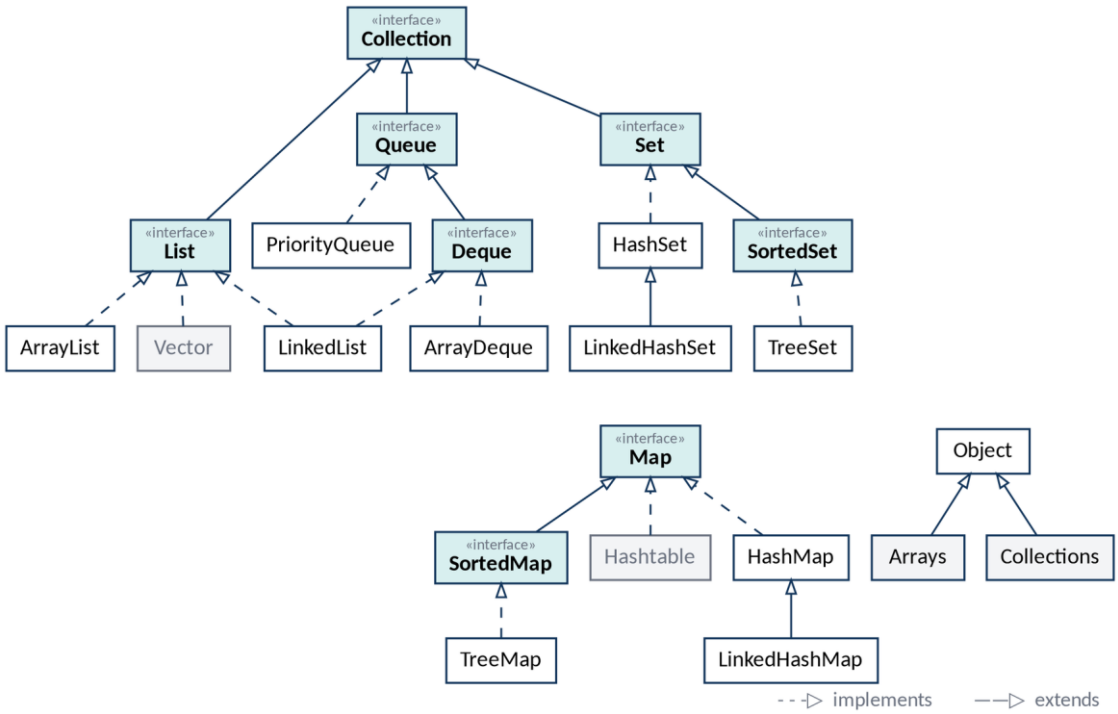


Figure 9.2 The core of Java's Collections Framework. Interfaces are marked «interface»; dashed arrows mean *implements* and solid arrows mean *extends*; the grey classes are historical. `Map` is not a `Collection`; `Arrays` and `Collections` hold static helper methods.

`Vector` and `Hashtable`, drawn in grey, are historical, older than the framework, and synchronize every operation, which makes them slower; new code uses `ArrayList` and `HashMap` (Chapter 14 shows the tools for sharing collections between threads). `Deque` (*double-ended queue*) is implemented by `ArrayDeque` and `LinkedList`, which is therefore both a `List` and a `Deque`. Two refinements are left out of the drawing to keep it readable: `EnumMap`/`EnumSet` for enum keys, and the three interfaces that Java 21 inserted into the hierarchy, `SequencedCollection`, `SequencedSet` and `SequencedMap`, which give every collection with a defined order the methods `getFirst`, `getLast`, `addFirst`, `addLast` and `reversed` [6]. `List`, `Deque`, `SortedSet` and `SortedMap` now extend them, and `LinkedHashSet` and `LinkedHashMap` implement them.

9.3.1 Three families: sets, lists, and maps

Almost every question about which collection to use starts by choosing among three families. **Table 9.1** sets them side by side.

Table 9.1 The three families of collections compared.

	Set	List	Map
Duplicates	none: adding an element that is already there changes nothing	allowed: the same element may appear many times	keys are unique; values may repeat
Order	not a concern (but `TreeSet` keeps sorted order and `LinkedHashSet` keeps insertion order)	the order of the elements is the point: every element has a position	not a concern (but `TreeMap` keeps keys sorted and `LinkedHashMap` keeps insertion order)
Main operations	`add`, `contains`, `remove`	`get(i)`, `set(i, x)`, `add`, `remove`	`get(key)`, `put(key, value)`
Everyday picture	the set of letters in a word: "one" and "neo" have the same letters	a sentence: "you and us us"	a dictionary: a word leads to its meaning
Python	`set`, `frozenset`	`list`, `tuple`	`dict`

A set answers *is this element in me?*; a list keeps everything, repeats included, where you put it; a map answers *what goes with this key?*, like a dictionary or *associative array*. Queues, the fourth family, are lists with a discipline for taking elements out (Section 9.7).

9.3.2 Program to the interface

The drawing has a practical consequence for how we declare variables. Write

```
List<String> myList = new ArrayList<String>();
```

rather than

```
ArrayList<String> myList = new ArrayList<String>();
```

The interface should be used as the variable's type whenever possible, because the interface is what the rest of the code relies on. If `myList` is declared as a `List`, the code that uses it can call only the operations that every list has. Later, if measurements show that a `LinkedList` would serve better, we change one word on the right-hand side, and nothing else in the program can break, because nothing else ever knew which class it was. Chapter 6 called this *programming to an interface*; with collections it is the everyday habit of Java programmers. The same applies to parameters: a method that accepts `List<String>` accepts every kind of list, while a method that demands `ArrayList<String>` needlessly rejects the others. (Since Java 7 the type arguments on the right may be left to the compiler, `new ArrayList<>()`: the *diamond*. We use it from now on.)

9.3.3 Python's collections

Python has fewer collection classes, and the most important ones are built into the language with their own syntax: `[1, 2]` makes a `list`, `(1, 2)` a `tuple`, `{1, 2}` a `set`, `{"WASH": 7000}` a `dict`. The module `collections` adds `deque`, `OrderedDict`, `defaultdict`, `Counter` and `namedtuple`; the modules `heapq` and `bisect` add algorithms that turn an ordinary list into a priority queue or a sorted array [7], [8].

Python's equivalent of Java's interfaces is the module `collections.abc`, whose abstract base classes name the kinds of collection: `Iterable` (can be looped over), `Sized` (has a `len`), `Container` (supports `in`), and their combination `Collection`; below that `Sequence`, `Set` and `Mapping`, each with a `Mutable…` version. **Figure 9.3** draws them. The built-in types do not inherit from these

classes in the ordinary way; they are *registered* as behaving like them, so that `isinstance([], collections.abc.MutableSequence)` is `True`. That is duck typing made official: a class that behaves like a sequence *is* a sequence.

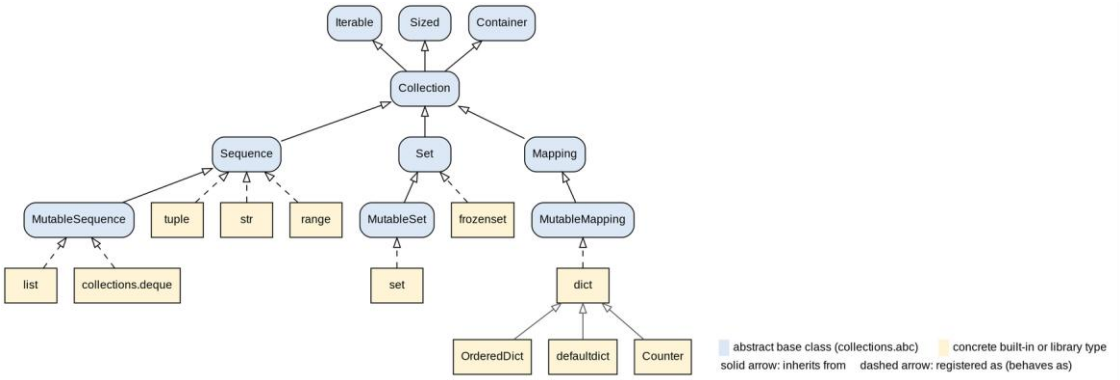


Figure 9.3 Python's collection types and the abstract base classes of `collections.abc`, the counterpart of Java's collection interfaces.

Table 9.2 pairs the two languages. Keep it at hand for the rest of the chapter.

Table 9.2 The collections of this chapter in Java and in Python.

Purpose	Java interface	Java classes	Python
sequence with positions	`List<E>`	`ArrayList`, `LinkedList`	`list`; `tuple` if it never changes
no duplicates, fast membership	`Set<E>`	`HashSet`, `LinkedHashSet`	`set`, `frozenset`
no duplicates, sorted	`SortedSet<E>`, `NavigableSet<E>`	`TreeSet`	`sorted(s)` when needed; `bisect` on a sorted list
key → value	`Map<K,V>`	`HashMap`, `LinkedHashMap`	`dict` (keeps insertion order since 3.7)
key → value, sorted keys	`SortedMap<K,V>`, `NavigableMap<K,V>`	`TreeMap`	`dict(sorted(d.items()))` when needed
first in, first out	`Queue<E>`	`ArrayDeque`, `LinkedList`	`collections.deque`
last in, first out	`Deque<E>`	`ArrayDeque`	`list` (`append`/`pop`)
smallest first	`Queue<E>`	`PriorityQueue`	`heapq` on a `list`
counting	`Map<K,Integer>` with `merge`	`HashMap`, `TreeMap`	`collections.Counter`
missing keys get a default	`computeIfAbsent`, `getOrDefault`	any `Map`	`collections.defaultdict`, `dict.get`

9.4 Lists

9.4.1 The everyday operations

We begin with the family we already know. **Listing 9.5** runs through the operations of `List`, first on a few words, then on positions, and finally passes the same list to one method as an `ArrayList` and as a `LinkedList`.

Listing 9.5 `ListDemo.java`: adding, testing, counting, bulk operations, positions, and one method that serves two list classes.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

public class ListDemo {
    public static void main(String[] args) {
        List<String> myList = new ArrayList<String>();   // interface on the left
        myList.add("you");
        myList.add("and");
        myList.add("us");
        System.out.println(myList);                        // [you, and, us]
        System.out.println(myList.contains("they"));       // false
        System.out.println(myList.add("us"));              // true: duplicates allowed
        System.out.println(myList.size());                 // 4

        String s = "";
        for (String w : myList) {
            s += w + " ";
        }
        System.out.println("Sentence = " + s);

        myList.addAll(Arrays.asList("love", "them"));
        System.out.println(myList);
        myList.removeAll(Arrays.asList("love", "them", "dude"));
        System.out.println(myList);
        myList.retainAll(Arrays.asList("and", "us", "love"));
        System.out.println(myList);

        // Positions: every element of a list has an index, starting at 0.
        myList.add(0, "we");           // insert at the front
        myList.set(1, "or");           // replace index 1
        System.out.println(myList + " get(2) = " + myList.get(2)
                + " indexOf(\"us\") = " + myList.indexOf("us"));

        // Programming to the interface: the same method serves both classes.
        describe(new ArrayList<>(myList));
        describe(new LinkedList<>(myList));
    }

    static void describe(List<String> words) {
        words.remove(words.size() - 1);                    // drop the last element
        System.out.println(words.getClass().getSimpleName() + " " + words
                + " size " + words.size());
    }
}
```

```
[you, and, us]
false
true
4
Sentence = you and us us
[you, and, us, us, love, them]
[you, and, us, us]
[and, us, us]
[we, or, us, us] get(2) = us indexOf("us") = 2
ArrayList [we, or, us] size 3
LinkedList [we, or, us] size 3
```

Follow the output line by line. After three `add` calls the list prints as `[you, and, us]`: a list's `toString` shows the elements in order, between brackets. `contains("they")` is `false`. `add("us")` returns `true` even though `"us"` is already there — a list allows duplicates, and `add` reports that the

list changed. The size is now 4, and the loop builds the sentence `"you and us us "` (note the space at the end, added after every word including the last).

The next three lines are the **bulk operations**, which take a whole collection as their argument. `addAll(Arrays.asList("love", "them"))` appends both words; `Arrays.asList` is the quick way to make a small fixed list from a few values. `removeAll(Arrays.asList("love", "them", "dude"))` removes every element that appears in the argument — `"dude"` is not in the list, which is harmless. `retainAll(Arrays.asList("and", "us", "love"))` does the opposite: it keeps only the elements that appear in the argument, so `"you"` goes and both copies of `"us"` stay: `[and, us, us]`. In the language of sets, `addAll` is a union, `removeAll` a difference and `retainAll` an intersection, but on a list they respect duplicates and order.

Then come positions. `add(0, "we")` inserts at the front and pushes everything else one place to the right; `set(1, "or")` replaces the element at index 1; `get(2)` reads index 2; `indexOf("us")` finds the *first* index holding `"us"`, here 2. Positions start at 0, as in arrays.

Finally, `describe(List<String> words)` is written against the interface. It is called once with a new `ArrayList` and once with a new `LinkedList`, each a copy of `myList` (every collection class has a constructor that copies another collection). The method removes the last element and prints the class name, the list and its size; the two lines differ only in the class name. That is programming to the interface in action.

Listing 9.6 does the same in Python.

Listing 9.6 `list_demo.py`: the same operations with Python's `list`, and one function that serves both `list` and `deque`.

```
"""list_demo.py: the everyday list operations, and deque as a second implementation."""
from collections import deque


def describe(words):
    words.pop()                                  # drop the last element
    print(type(words).__name__, list(words), "size", len(words))


def main():
    my_list: list[str] = []
    my_list.append("you")
    my_list.append("and")
    my_list.append("us")
    print(my_list)                       # ['you', 'and', 'us']
    print("they" in my_list)             # False
    my_list.append("us")                 # append returns None; duplicates allowed
    print(len(my_list))                  # 4

    s = ""
    for w in my_list:
        s += w + " "
    print("Sentence = " + s)
    print("Sentence = " + " ".join(my_list))     # the Pythonic way, no trailing space

    my_list.extend(["love", "them"])
    print(my_list)
    my_list = [w for w in my_list if w not in {"love", "them", "dude"}]   # removeAll
    print(my_list)
    my_list = [w for w in my_list if w in {"and", "us", "love"}]          # retainAll
    print(my_list)

    # Positions: every element of a list has an index, starting at 0.
    my_list.insert(0, "we")              # insert at the front
    my_list[1] = "or"                    # replace index 1
    print(my_list, "get(2) =", my_list[2], 'indexOf("us") =', my_list.index("us"))

    # Duck typing: the same function serves both classes.
    describe(list(my_list))
    describe(deque(my_list))


if __name__ == "__main__":
    main()
```

```
['you', 'and', 'us']
False
4
Sentence = you and us us
Sentence = you and us us
['you', 'and', 'us', 'us', 'love', 'them']
['you', 'and', 'us', 'us']
['and', 'us', 'us']
['we', 'or', 'us', 'us'] get(2) = us indexOf("us") = 2
list ['we', 'or', 'us'] size 3
deque ['we', 'or', 'us'] size 3
```

The correspondence is close. `append`, `insert`, indexing, `index`, `len` and `in` do the jobs of `add`, `add(i, x)`, `get`/`set`, `indexOf`, `size` and `contains`; `append` returns `None`, because a Python list never refuses an element. Python has no `removeAll` or `retainAll`: the Pythonic way is a **list comprehension**, `[w for w in my_list if w not in {"love", "them", "dude"}]`, which builds a *new* list of the elements that pass the test (write `my_list[:] = [...]` to change the original object in place). `" ".join(my_list)` builds the sentence without the trailing space. The hint `my_list: list[str]` is for tools such as mypy; Python ignores it when running (Section

9.8.7). And `describe` serves a `list` and a `collections.deque` with no declared interface, because both have `pop` and `len`: duck typing, Python's way of programming to an interface.

9.4.2 ArrayList versus LinkedList

Both classes implement `List`, so they offer the same operations. They store their elements completely differently, and that makes the same operation cheap in one and expensive in the other. **Figure 9.4** shows the list `[we, or, us, us]` in both forms.

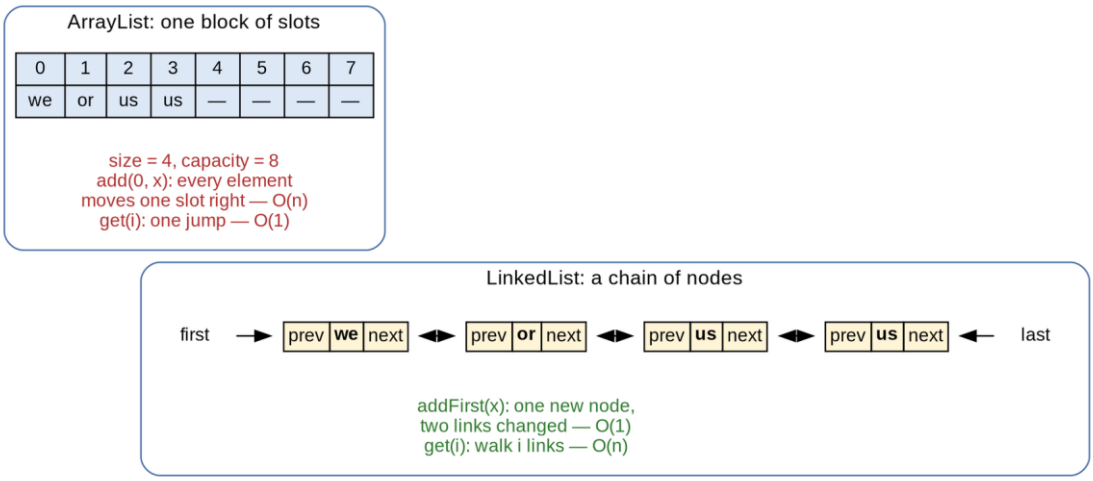


Figure 9.4 The same list stored two ways. An `ArrayList` keeps references in one block of consecutive slots, with spare capacity at the end; a `LinkedList` keeps a chain of nodes, each linked to its neighbors.

An `ArrayList` is the growing array of Chapter 5: one block of consecutive slots, of which the first `size` are in use. Because the slots are consecutive, the address of slot i is computed in one step, so `get(i)` and `set(i, x)` take constant time, $O(1)$. Adding at the end is also $O(1)$ on average (the occasional regrowth is paid for by the many cheap adds, as Chapter 5 showed). But inserting or removing at the *front* or in the middle forces every later element to move one slot. Inserting n elements one after another at the front of an initially empty `ArrayList` therefore moves

$$0 + 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} \tag{9.3}$$

elements in total, about 5×10^9 moves for $n = 100{,}000$.

A `LinkedList` is a chain of small node objects, each holding one element and two references, `prev` and `next`, plus references to the `first` and `last` node. Inserting at either end creates one node and changes two references: $O(1)$, however long the list. Removing through an iterator that is already standing at the right place is also $O(1)$. But there is no way to jump to position i: `get(i)` must walk the chain from the nearer end, up to $n/2$ steps, so it costs $O(n)$. A loop `for (int i = 0; i < list.size(); i++) list.get(i)` over a `LinkedList` therefore costs $O(n^2)$ in total — a well-known way to make a fast program slow.

In practice `ArrayList` almost always wins: its slots sit next to each other in memory, which processors read very fast, while every `LinkedList` node is a separate object scattered across memory. For cheap work at *both ends*, `ArrayDeque` (Section 9.7) beats `LinkedList` too. Python's `list` is an `ArrayList`; its `deque` covers the both-ends case.

9.4.3 Visiting every element, and removing safely

There are three ways to visit the elements of a list, and one famous trap. **Listing 9.7** shows all four.

Listing 9.7 `IterationDemo.java`: three ways to visit a list; removing inside a for-each loop fails, while `Iterator.remove` and `removeIf` succeed.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class IterationDemo {
    public static void main(String[] args) {
        List<String> words = new ArrayList<>(List.of("you", "us", "us", "and", "us"));

        // 1. Three ways to visit every element.
        for (int i = 0; i < words.size(); i++) {       // by index
            System.out.print(words.get(i) + " ");
        }
        System.out.println();
        for (String w : words) {                        // for-each (uses an Iterator)
            System.out.print(w + " ");
        }
        System.out.println();
        Iterator<String> it = words.iterator();         // the Iterator by hand
        while (it.hasNext()) {
            System.out.print(it.next() + " ");
        }
        System.out.println();

        // 2. Removing inside a for-each loop: the list changes under the iterator.
        try {
            for (String w : words) {
                if (w.equals("us")) {
                    words.remove(w);
                }
            }
        } catch (java.util.ConcurrentModificationException e) {
            System.out.println("for-each + remove: " + e.getClass().getSimpleName());
        }
        System.out.println("words now: " + words);

        // 3. Removing through the iterator itself is safe.
        words = new ArrayList<>(List.of("you", "us", "us", "and", "us"));
        Iterator<String> it2 = words.iterator();
        while (it2.hasNext()) {
            if (it2.next().equals("us")) {
                it2.remove();
            }
        }
        System.out.println("iterator.remove: " + words);
```

Listing 9.7 (Continued) *IterationDemo.java: three ways to visit a list; removing inside a for-each loop fails, while Iterator.remove and removeIf succeed.*

```
        // 4. Or say what to remove, and let the list do the loop.
        words = new ArrayList<>(List.of("you", "us", "us", "and", "us"));
        words.removeIf(w -> w.equals("us"));
        System.out.println("removeIf: " + words);
    }
}
```

```
you us us and us
you us us and us
you us us and us
for-each + remove: ConcurrentModificationException
words now: [you, us, and, us]
iterator.remove: [you, and]
removeIf: [you, and]
```

The first three lines print the same words in the same order. The index loop uses `get(i)`, which is fine for an `ArrayList` and slow for a `LinkedList`. The **for-each loop**, `for (String w : words)`, works for every collection, because every collection is `Iterable`: it has a method `iterator()` that returns an **`Iterator`**, an object that walks the collection one element at a time. The third loop does by hand what for-each does behind the scenes: `hasNext()` asks whether another element remains, and `next()` returns it and moves on. An iterator only moves forward — there is no `prev()` — and it reads the collection; it does not copy it.

The second block is the trap. Inside a for-each loop we call `words.remove(w)` for every `"us"`. The first `"us"` is found and removed. On the next turn of the loop, the hidden iterator notices that the list was changed behind its back — every `ArrayList` counts its structural changes, and the iterator compares that count with the one it remembered — and throws `ConcurrentModificationException`. The list is left half-cleaned, `[you, us, and, us]`. The name is misleading: no threads are involved; "concurrent" means that the list was modified *while* an iteration was in progress. This behavior is called **fail-fast**, and it is a kindness: the error appears at once, near its cause.

There are two correct ways. The iterator itself has a `remove()` method that removes the element most recently returned by `next()` and keeps the iterator's own records up to date, so the third block removes all three `"us"` safely. Simplest of all, since Java 8 every collection has `removeIf`, which takes a *condition* and does the loop internally. The condition `w -> w.equals("us")` is a **lambda expression**: a small function written inline, here "given `w`, answer whether `w` equals `"us"`". Chapter 6 used one with `assertThrows`, and Chapter 10 explains lambdas fully; for now, read the arrow as "such that".

Python has the same trap, in a nastier form, as **Listing 9.8** shows.

Listing 9.8 `iteration_demo.py`: the same walks in Python. Removing from a list inside a `for` loop silently skips elements; changing a dictionary's size inside a loop raises `RuntimeError`.

```
"""iteration_demo.py: visiting a list, and removing from it safely."""


def main():
    words = ["you", "us", "us", "and", "us"]

    # 1. Three ways to visit every element.
    for i in range(len(words)):                 # by index
        print(words[i], end=" ")
    print()
    for w in words:                             # for (uses an iterator)
        print(w, end=" ")
    print()
    it = iter(words)                            # the iterator by hand
    while True:
        try:
            print(next(it), end=" ")
        except StopIteration:
            break
    print()

    # 2. Removing inside a for loop: no exception, but elements are skipped!
    for w in words:
        if w == "us":
            words.remove(w)
    print("for + remove: no error")
    print("words now:", words)

    # 3. Loop over a copy, change the original.
    words = ["you", "us", "us", "and", "us"]
    for w in words[:]:
        if w == "us":
            words.remove(w)
    print("copy + remove:", words)

    # 4. Or build the list you want in one expression.
    words = ["you", "us", "us", "and", "us"]
    words = [w for w in words if w != "us"]
    print("comprehension:", words)
```

Listing 9.8 (Continued) *iteration_demo.py: the same walks in Python. Removing from a list inside a for loop silently skips elements; changing a dictionary's size inside a loop raises RuntimeError.*

```
    # A dict, unlike a list, does notice the change.
    stock = {"soap": 3, "bleach": 0}
    try:
        for item in stock:
            if stock[item] == 0:
                del stock[item]
    except RuntimeError as e:
        print("RuntimeError:", e)


if __name__ == "__main__":
    main()
```

```
you us us and us
you us us and us
you us us and us
for + remove: no error
words now: ['you', 'and', 'us']
copy + remove: ['you', 'and']
comprehension: ['you', 'and']
RuntimeError: dictionary changed size during iteration
```

The hand-driven loop uses the built-in functions `iter` and `next`; when the iterator is exhausted, `next` raises `StopIteration`, which is how every Python `for` loop knows when to stop. The second block removes from the list while looping over it, and Python raises **no error** — but the result `['you', 'and', 'us']` still contains an `"us"`. Trace it. The list starts as `[you, us, us, and, us]`. The loop's hidden iterator stands at index 1 (`"us"`), and `remove` deletes it; the list closes the gap, so the second `"us"` moves into index 1, and the iterator, moving on to index 2, never sees it. At index 3 the loop meets the last `"us"` and calls `words.remove("us")` — but `remove` deletes the *first* `"us"` in the list, the one the loop skipped, at index 1. The loop then runs off the end, and the last `"us"` survives. A bug that gives a *plausible* wrong answer is the worst kind. The cures are to loop over a copy, `words[:]`, while changing the original, or — better — to build the list you want with a comprehension. Dictionaries and sets are fail-fast like Java's collections: changing their size during a loop raises `RuntimeError: dictionary changed size during iteration`, as the last block shows.

> ✓ **KEY INSIGHT.** Never change the *structure* of a collection (add or remove elements) while a loop is walking it, except through the walker itself (`Iterator.remove`). Either collect the changes and apply them afterwards, or build a new collection. Replacing an element in place (`set(i, x)`, `words[i] = x`) is harmless, because the structure does not change.

9.5 Sets

9.5.1 A set of numbers, and a set of names

A **set** is a collection without duplicates. Its central question is membership — *is this element in me?* — and it answers that question far faster than a list does. **Listing 9.9** starts with a `TreeSet` of integers and its set algebra, then puts the same six arrivals into three kinds of set.

Listing 9.9 `SetDemo.java`: a `TreeSet` of numbers with union, difference and intersection; the same names in a `HashSet`, a `LinkedHashSet` and a `TreeSet`; nearest-element queries; and sets of letters.

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;

public class SetDemo {
    public static void main(String[] args) {
        Set<Integer> myNumbers = new TreeSet<Integer>();
        myNumbers.add(1);
        myNumbers.add(2);
        myNumbers.add(3);
        System.out.println(myNumbers);                    // [1, 2, 3]
        System.out.println(myNumbers.contains(7));        // false
        System.out.println(myNumbers.add(3));             // false: already there
        System.out.println(myNumbers.size());             // 3

        int sum = 0;
        for (int n : myNumbers) {
            sum += n;
        }
        System.out.println("Sum = " + sum);               // Sum = 6

        myNumbers.addAll(Arrays.asList(1, 2, 3, 4, 5, 6, 7));
        System.out.println(myNumbers);                    // union
        myNumbers.removeAll(Arrays.asList(4, 5, 6, 7, 8, 9, 10));
        System.out.println(myNumbers);                    // difference
        myNumbers.retainAll(Arrays.asList(2, 3, 4, 5));
        System.out.println(myNumbers);                    // intersection
        System.out.println();

        // The same words in three kinds of set.
        List<String> arrivals = List.of("Sari", "Budi", "Rina", "Agus", "Budi", "Sari");
        System.out.println("HashSet       " + new HashSet<>(arrivals));
        System.out.println("LinkedHashSet " + new LinkedHashSet<>(arrivals));
        System.out.println("TreeSet       " + new TreeSet<>(arrivals));

        // A TreeSet knows its order, so it can answer "nearest" questions.
        TreeSet<String> names = new TreeSet<>(arrivals);
        System.out.println("first " + names.first() + ", last " + names.last()
                + ", ceiling(\"C\") " + names.ceiling("C")
                + ", headSet(\"R\") " + names.headSet("R"));
```

Listing 9.9 (Continued) *SetDemo.java: a TreeSet of numbers with union, difference and intersection; the same names in a HashSet, a LinkedHashSet and a TreeSet; nearest-element queries; and sets of letters.*

```
        // Order does not matter to a set: the letters of "one" and "neo" are the same set.
        System.out.println("letters(one).equals(letters(neo)) = "
                + letters("one").equals(letters("neo")));
    }

    static Set<Character> letters(String word) {
        Set<Character> result = new TreeSet<>();
        for (char c : word.toCharArray()) {
            result.add(c);
        }
        return result;
    }
}
```

```
[1, 2, 3]
false
false
3
Sum = 6
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3]
[2, 3]

HashSet       [Sari, Budi, Agus, Rina]
LinkedHashSet [Sari, Budi, Rina, Agus]
TreeSet       [Agus, Budi, Rina, Sari]
first Agus, last Sari, ceiling("C") Rina, headSet("R") [Agus, Budi]
letters(one).equals(letters(neo)) = true
```

The first block mirrors the list example, and the differences are the point. `add(3)` on a set that already holds 3 returns `false`: a set holds each element at most once. The bulk operations are now true set algebra: `addAll` forms the **union** $\{1,2,3\} \cup \{1,\dots,7\} = \{1,\dots,7\}$, `removeAll` the **difference** $\{1,\dots,7\} \setminus \{4,\dots,10\} = \{1,2,3\}$, and `retainAll` the **intersection** $\{1,2,3\} \cap \{2,3,4,5\} = \{2,3\}$. A `TreeSet` always prints in increasing order.

The second block is the most instructive four lines of the section. Six arrivals, with Budi and Sari arriving twice, become four distinct names in each set — but in three different orders:

- The `HashSet` prints `[Sari, Budi, Agus, Rina]`, an order that looks random. It is not random; it is the order of the *hash buckets*, explained in the next subsection. It is, however, an order you must never rely on: it can change when the set grows or between Java versions.
- The `LinkedHashSet` prints `[Sari, Budi, Rina, Agus]`, the order of first arrival. It is a hash set that also threads its elements on a linked list in insertion order, at the cost of two extra references per element.
- The `TreeSet` prints `[Agus, Budi, Rina, Sari]`, sorted. It keeps its elements in a balanced search tree, which also lets it answer questions a hash set cannot: `first()` and `last()`, `ceiling("C")` (the smallest element $\geq$ `"C"`: no name in the set starts with C to Q, so the answer is `Rina`), and `headSet("R")` (all elements $<$ `"R"`).

The last line shows why a set suits the question “which letters does this word use?”. The words “one” and “neo” contain the same letters in a different order, so their letter sets are equal. (Sets are not enough to detect *anagrams*, words that use the same letters the same number of times: “aab” and “abb” have equal letter sets but are not anagrams. Exercise 9.4 builds a correct anagram finder with a map.)

Listing 9.10 is the Python version.

Listing 9.10 `set_demo.py`: Python’s `set` with its operators, the orders Java’s three sets provide, binary search on a sorted list, and a `frozenset`.

```
"""set_demo.py: Python's set, and how to get the orders Java's three sets give."""
from bisect import bisect_left


def main():
    my_numbers: set[int] = set()
    my_numbers.add(1)
    my_numbers.add(2)
    my_numbers.add(3)
    print(my_numbers)                           # {1, 2, 3}
    print(7 in my_numbers)                      # False
    before = len(my_numbers)
    my_numbers.add(3)                           # add returns None ...
    print(len(my_numbers) > before)             # ... so ask whether the size grew
    print(len(my_numbers))                      # 3

    total = 0
    for n in my_numbers:
        total += n
    print("Sum =", total)                       # or simply sum(my_numbers)

    my_numbers |= {1, 2, 3, 4, 5, 6, 7}         # union
    print(my_numbers)
    my_numbers -= {4, 5, 6, 7, 8, 9, 10}        # difference
    print(my_numbers)
    my_numbers &= {2, 3, 4, 5}                  # intersection
    print(my_numbers)
    print({1, 2, 3} ^ {2, 3, 4}, "(symmetric difference)")
    print()

    arrivals = ["Sari", "Budi", "Rina", "Agus", "Budi", "Sari"]
    distinct = set(arrivals)                    # iteration order may change from run to run
    print("set size   ", len(distinct))
    print("first-seen ", list(dict.fromkeys(arrivals)))   # like LinkedHashSet
    print("sorted     ", sorted(distinct))                 # like TreeSet

    # "Nearest" questions on a sorted list, with binary search.
    names = sorted(distinct)
    i = bisect_left(names, "C")
    print(f"first {names[0]}, last {names[-1]}, ceiling(\"C\") {names[i]}, "
          f"headSet(\"R\") {names[:bisect_left(names, 'R')]}")

    print("set('one') == set('neo') =", set("one") == set("neo"))
    frozen = frozenset({"WASH", "IRON"})        # an immutable set: usable as a dict key
    print("frozenset is hashable:", hash(frozen) == hash(frozenset({"IRON", "WASH"})))


if __name__ == "__main__":
    main()
```

Listing 9.10 (Continued) *set_demo.py: Python's set with its operators, the orders Java's three sets provide, binary search on a sorted list, and a frozenset.*

```
{1, 2, 3}
False
False
3
Sum = 6
{1, 2, 3, 4, 5, 6, 7}
{1, 2, 3}
{2, 3}
{1, 4} (symmetric difference)

set size    4
first-seen  ['Sari', 'Budi', 'Rina', 'Agus']
sorted      ['Agus', 'Budi', 'Rina', 'Sari']
first Agus, last Sari, ceiling("C") Rina, headSet("R") ['Agus', 'Budi']
set('one') == set('neo') = True
frozenset is hashable: True
```

Python writes set algebra with operators: `|` for union, `-` for difference, `&` for intersection and `^` for symmetric difference (the elements in exactly one of the two sets), each with an in-place form such as `|=`. `add` returns `None`, so the listing compares the size before and after to learn whether the element was new.

Notice what the listing does *not* print: the set of names itself. A Python `set` of strings iterates in an order that changes *from one run to the next*, because Python salts string hashes with a random value at start-up, a defense against data crafted to collide. Integers are not salted, so `{1, 2, 3}` prints predictably. When order matters, ask for it: `sorted(s)` gives Java's `TreeSet` order, and `list(dict.fromkeys(items))` gives `LinkedHashSet`'s first-seen order (a `dict` remembers insertion order, and `fromkeys` makes a dictionary whose keys are the items). For nearest-element questions, keep a sorted list and use the module `bisect`: `bisect_left(names, "C")` is the index at which `"C"` would be inserted to keep the list sorted, found by binary search, so `names[i]` is the ceiling. Finally, a `frozenset` is an immutable set; being unchangeable, it has a stable hash and can be an element of another set or a key of a dictionary.

9.5.2 How a HashSet finds an element: hashing

Why is `contains` on a `HashSet` fast? Because the set does not search. It *computes* where the element must be.

Every Java object has a method `hashCode()` returning an `int`, and Chapters 6 and 7 insisted that equal objects must return equal hash codes. For a `String` s of length k with characters $s_0, \ldots, s_{k-1}$ the code is $s_0 \cdot 31^{k-1} + s_1 \cdot 31^{k-2} + \cdots + s_{k-1}$, computed with 32-bit wrap-around. For `"Sari"` that is $83 \cdot 31^3 + 97 \cdot 31^2 + 114 \cdot 31 + 105 = 2{,}472{,}653 + 93{,}217 + 3{,}534 + 105 = 2{,}569{,}509$.

A `HashSet` (really a `HashMap` inside, with the elements as keys) keeps an array of m **buckets**; a new set starts with $m = 16$. To place an element with hash code h, it mixes the high bits into the low bits and keeps the lowest four bits, `(h ^ (h >>> 16)) & 15`, a number from 0 to 15. For `"Sari"` this gives bucket 2. **Figure 9.5** shows all four names.

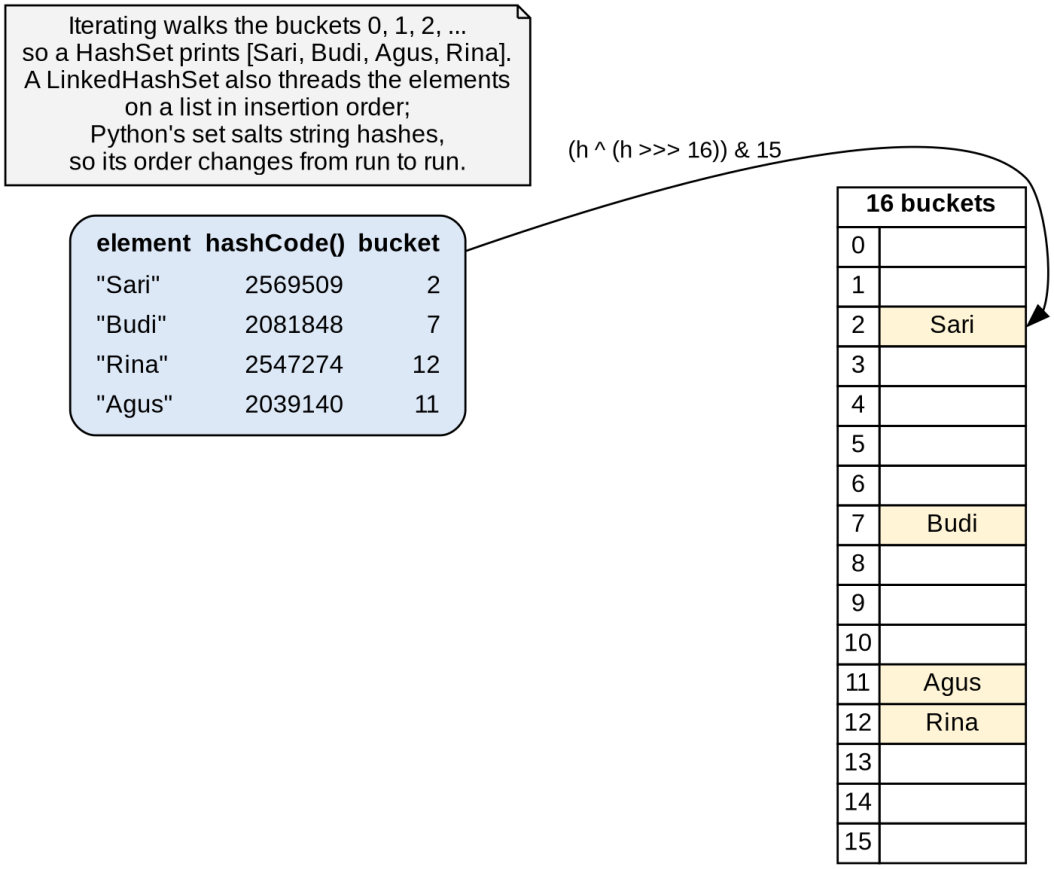


Figure 9.5 A `HashSet` of four names. Each name's hash code selects one of 16 buckets, so `contains` looks in one bucket only; iteration walks the buckets in index order, which explains the printed order `[Sari, Budi, Agus, Rina]`.

`contains("Rina")` computes the bucket, 12, and compares `"Rina"` only with what lies in bucket 12, using `equals`. It never looks at the other names. Iteration walks the buckets from 0 to 15, which is exactly why **Listing 9.9** printed `[Sari, Budi, Agus, Rina]`: buckets 2, 7, 11, 12.

Two different elements may land in the same bucket, a **collision**; the bucket then holds a short chain of elements, and `contains` compares against each of them. How long are the chains? If n elements are spread over m buckets, the average chain has length equal to the **load factor**

$$\alpha = \frac{n}{m} \tag{9.4}$$

and, assuming a hash function that spreads elements evenly, an unsuccessful search examines on average α elements after computing the bucket, and a successful one about $1 + \alpha/2$ [9]. Java keeps α small by force: when n exceeds $0.75\,m$, the table doubles its number of buckets and redistributes every element. So $\alpha \le 0.75$ always holds, and the expected cost of `add`, `contains` and `remove` is

$$T_{\text{hash}}(n) = O(1 + \alpha) = O(1) \tag{9.5}$$

— constant, however many elements the set holds. The doubling costs $O(n)$ when it happens, but, like the growing array of Chapter 5, so rarely that the average cost per insertion stays constant. (Since Java 8 a bucket with more than eight colliding elements becomes a small tree, which limits the worst case to $O(\log n)$. Python's `set` and `dict` resolve collisions by *open addressing*, trying another slot of the same array, with the same expected $O(1)$.)

All of this rests on `hashCode` and `equals` being consistent — equal objects, equal hash codes — and on the hash code *not changing* while the object sits in the set. Exercise 9.8 shows what happens when it does change.

9.5.3 How a TreeSet keeps order

A `TreeSet` keeps its elements in a **binary search tree**: every node has at most two children, and every element in a node's left subtree is smaller than the node's element, every element in its right subtree larger. **Figure 9.6** shows the four names.

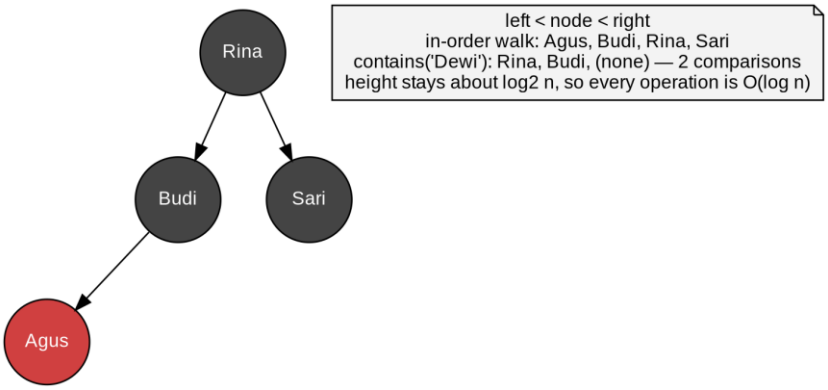


Figure 9.6 The `TreeSet` of the same four names: a red-black search tree. An in-order walk visits the names in sorted order; a search follows one path from the root.

To find `"Dewi"`, compare with the root `Rina`: Dewi comes before Rina, so go left, to `Budi`; Dewi comes after Budi, so go right — there is nothing there, so `"Dewi"` is absent. Two comparisons. Walking the tree *in order* (left subtree, node, right subtree, the traversal of Problem 8.7) visits the elements sorted, which is how a `TreeSet` prints and iterates.

A search tree is fast only if it is *balanced*: a tree built by inserting already-sorted keys into a plain search tree degenerates into a chain, and searching it costs $O(n)$, as Solution 7.2 observed. Java's `TreeSet` and `TreeMap` use a **red-black tree**, whose nodes are colored so that no path from the root to a leaf is more than twice as long as any other [10]. This guarantees that a tree of n elements has height

$$h \leq 2\log_2(n+1) \tag{9.6}$$

so `add`, `contains` and `remove` visit $O(\log n)$ nodes [9]. For a million elements that is at most about 40 comparisons, against about 500,000 for a list. A tree set is slower than a hash set by a factor of 10 or 20, and in exchange it keeps order and answers range questions (`first`, `ceiling`, `headSet`, `subSet`). Its elements must be comparable: they must implement `Comparable`, or the set must be given a `Comparator` when it is created (Section 9.9). Python has no tree-based set in its standard library; `sorted()` and `bisect` on a list cover most needs, and the third-party package *sortedcontainers* covers the rest.

9.6 Maps

9.6.1 A map from keys to values

A **map** associates each *key* with one *value*. Keys are unique, like the words of a dictionary; values may repeat. A map is the collection behind every "look it up" question: order number → order, service code → price, customer → list of orders, word → number of occurrences.

Listing 9.11 begins with the style in which much older Java code is written: *raw types*, without type arguments, so that every value comes out as an `Object` and must be cast. Then it does the same job with generics and meets the most useful `Map` methods.

Listing 9.11 `MapDemo.java`: walking a map's entries, first with raw types and casts, then with generics; the main `Map` methods on a price list; counting with `merge`; and three kinds of map.

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;

public class MapDemo {
    @SuppressWarnings({"rawtypes", "unchecked"})
    static void oldStyle() {
        Map<Object, String> map = new HashMap<>();
        map.put(1, "you");
        map.put(2, "and");
        map.put(3, "us");

        Set info = map.entrySet();                    // raw types: no <...>
        Iterator iterator = info.iterator();
        while (iterator.hasNext()) {
            Map.Entry m = (Map.Entry) iterator.next(); // every step needs a cast
            int key = (Integer) m.getKey();
            String value = (String) m.getValue();
            System.out.println("key: " + key + " -> value: " + value);
        }
        System.out.println();
        map.remove(2);
        info = map.entrySet();
        iterator = info.iterator();
        while (iterator.hasNext()) {
            Map.Entry m = (Map.Entry) iterator.next();
            int key = (Integer) m.getKey();
            String value = (String) m.getValue();
            System.out.println("key: " + key + " -> value: " + value);
        }
        System.out.println();
    }

    public static void main(String[] args) {
        oldStyle();
```

Listing 9.11 (Continued) *MapDemo.java: walking a map's entries, first with raw types and casts, then with generics; the main Map methods on a price list; counting with merge; and three kinds of map.*

```
        // The same walk with generics: no casts at all.
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "you");
        map.put(2, "and");
        map.put(3, "us");
        map.remove(2);
        for (Map.Entry<Integer, String> e : map.entrySet()) {
            System.out.println("key: " + e.getKey() + " -> value: " + e.getValue());
        }

        // A price list: service code -> rupiah per kilogram.
        Map<String, Integer> rates = new HashMap<>();
        System.out.println("put WASH: " + rates.put("WASH", 7000));       // null: new key
        rates.put("IRON", 5000);
        rates.put("WASH_IRON", 10000);
        System.out.println("put WASH: " + rates.put("WASH", 7500));       // 7000: old
value
        System.out.println("get IRON = " + rates.get("IRON")
                + ", get SHOES = " + rates.get("SHOES")
                + ", getOrDefault SHOES = " + rates.getOrDefault("SHOES", 0));
        System.out.println("containsKey(\"IRON\") = " + rates.containsKey("IRON")
                + ", containsValue(5000) = " + rates.containsValue(5000));
        System.out.println("keys " + new TreeMap<>(rates).keySet()
                + ", values " + new TreeMap<>(rates).values());

        // Counting with merge: add 1, or start at 1 if the key is new.
        String[] tickets = {"WASH", "IRON", "WASH", "WASH_IRON", "WASH", "IRON"};
        Map<String, Integer> counts = new LinkedHashMap<>();
        for (String t : tickets) {
            counts.merge(t, 1, Integer::sum);
        }
        System.out.println("LinkedHashMap " + counts);
        System.out.println("TreeMap       " + new TreeMap<>(counts));
        map.forEach((k, v) -> System.out.println("forEach " + k + " = " + v));
    }
}
```

```
key: 1 -> value: you
key: 2 -> value: and
key: 3 -> value: us

key: 1 -> value: you
key: 3 -> value: us

key: 1 -> value: you
key: 3 -> value: us
put WASH: null
put WASH: 7000
get IRON = 5000, get SHOES = null, getOrDefault SHOES = 0
containsKey("IRON") = true, containsValue(5000) = true
keys [IRON, WASH, WASH_IRON], values [5000, 7500, 10000]
LinkedHashMap {WASH=3, IRON=2, WASH_IRON=1}
TreeMap       {IRON=2, WASH=3, WASH_IRON=1}
forEach 1 = you
forEach 3 = us
```

A map is not a collection, so `oldStyle` walks it through `entrySet()`, a *view* of the map as a `Set` of `Map.Entry` objects, each holding one key and one value. With raw types every `next()` returns an `Object`, so the code casts to `Map.Entry`, then the key to `Integer` and the value to `String` — three promises the compiler cannot check. `@SuppressWarnings({"rawtypes", "unchecked"})` silences the warnings for exactly this code. After `map.remove(2)` only keys 1 and 3 remain. (Small `Integer`

keys happen to come out of a `HashMap` in increasing order, because a small integer's hash code is the integer itself; for sorted keys, still ask for a `TreeMap`.)

The generic version in `main` needs no casts at all: `Map<Integer, String>` tells the compiler the types, `Map.Entry<Integer, String> e` receives each entry, and `e.getKey()` is already an `Integer`. Section 9.8 explains how.

The price list shows the everyday methods:

- `put(key, value)` stores a value and **returns the previous value** for that key, or `null` if there was none. The first `put("WASH", 7000)` prints `null`; the second, `put("WASH", 7500)`, replaces it and returns the old 7000. A map never holds two values for one key.
- `get(key)` returns the value, or `null` if the key is absent; `getOrDefault(key, d)` returns `d` instead of `null`.
- `containsKey` is fast (a hash lookup); `containsValue` must look at every value, $O(n)$.
- `keySet()` and `values()` are views of the keys and the values. Wrapping the map in a `TreeMap` makes them come out sorted.

Counting is the most common use of a map, and `merge` does it in one line: `counts.merge(t, 1, Integer::sum)` means "if `t` is absent, store 1; otherwise store the old value plus 1". `Integer::sum` is a *method reference*, a short way of passing the existing method `Integer.sum(a, b)` as the rule for combining old and new values. The `LinkedHashMap` prints the counts in the order the codes first appeared, the `TreeMap` in alphabetical order of the keys. Finally, `map.forEach((k, v) -> ...)` visits every entry with a two-parameter lambda.

Listing 9.12 shows Python's `dict`.

Listing 9.12 `map_demo.py`: Python's `dict`, its `get` with a default, `KeyError`, and three ways to count: `get`, `Counter` and `defaultdict`.

```
"""map_demo.py: Python's dict, the counterpart of Java's Map."""
from collections import Counter, defaultdict


def main():
    d = {}
    d[1] = "you"
    d[2] = "and"
    d[3] = "us"
    for key, value in d.items():
        print(f"key: {key} -> value: {value}")
    print()
    del d[2]
    for key, value in d.items():
        print(f"key: {key} -> value: {value}")
    print()

    # A price list: service code -> rupiah per kilogram.
    rates: dict[str, int] = {}
    print("put WASH:", rates.get("WASH"))            # None: new key
    rates["WASH"] = 7000
    rates["IRON"] = 5000
    rates["WASH_IRON"] = 10000
    old = rates.get("WASH")
    rates["WASH"] = 7500
    print("put WASH:", old)                          # 7000: old value
    print(f"get IRON = {rates['IRON']}, get SHOES = {rates.get('SHOES')}, "
          f"getOrDefault SHOES = {rates.get('SHOES', 0)}")
    print(f'containsKey("IRON") = {"IRON" in rates}, '
          f"containsValue(5000) = {5000 in rates.values()}")
    print(f"keys {sorted(rates)}, values {[rates[k] for k in sorted(rates)]}")
    try:
        rates["SHOES"]
    except KeyError as e:
        print("rates['SHOES'] -> KeyError:", e)
```

Listing 9.12 (Continued) *map_demo.py: Python's dict, its get with a default, KeyError, and three ways to count: get, Counter and defaultdict.*

```
    # Counting: three ways.
    tickets = ["WASH", "IRON", "WASH", "WASH_IRON", "WASH", "IRON"]
    counts: dict[str, int] = {}
    for t in tickets:
        counts[t] = counts.get(t, 0) + 1
    print("dict        ", counts)                    # insertion order, like LinkedHashMap
    print("sorted      ", dict(sorted(counts.items())))  # like TreeMap
    print("Counter     ", Counter(tickets).most_common(2))
    by_first_letter = defaultdict(list)              # missing keys start as []
    for t in dict.fromkeys(tickets):
        by_first_letter[t[0]].append(t)
    print("defaultdict ", dict(by_first_letter))


if __name__ == "__main__":
    main()
```

```
key: 1 -> value: you
key: 2 -> value: and
key: 3 -> value: us

key: 1 -> value: you
key: 3 -> value: us

put WASH: None
put WASH: 7000
get IRON = 5000, get SHOES = None, getOrDefault SHOES = 0
containsKey("IRON") = True, containsValue(5000) = True
keys ['IRON', 'WASH', 'WASH_IRON'], values [5000, 7500, 10000]
rates['SHOES'] -> KeyError: 'SHOES'
dict         {'WASH': 3, 'IRON': 2, 'WASH_IRON': 1}
sorted       {'IRON': 2, 'WASH': 3, 'WASH_IRON': 1}
Counter      [('WASH', 3), ('IRON', 2)]
defaultdict  {'W': ['WASH', 'WASH_IRON'], 'I': ['IRON']}
```

A `dict` is written with braces and colons and indexed with square brackets. `d.items()` gives the key–value pairs, which the `for` loop unpacks into `key, value`, with no casts and no entry objects in sight. Two behaviors differ from Java and matter every day:

- `d[key]` on a missing key raises `KeyError` instead of returning `None`. Use `d.get(key)`, which returns `None`, or `d.get(key, default)`. Assignment `d[key] = value` returns nothing, so the listing reads the old value with `get` before replacing it.
- A `dict` remembers **insertion order** (guaranteed since Python 3.7), so it behaves like a `LinkedHashMap` by default. For sorted order, `dict(sorted(counts.items()))` builds a new dictionary from the pairs sorted by key.

For counting, `counts[t] = counts.get(t, 0) + 1` is the direct translation of `merge`. The class `collections.Counter` does it all at once and adds `most_common(k)`. And `collections.defaultdict(list)` is a dictionary that creates a missing value by calling `list()` the first time a key is used, so `by_first_letter[t[0]].append(t)` needs no "is the key already there?" test. That is the Python counterpart of Java's `computeIfAbsent(key, k -> new ArrayList<>())`, which the laundry order book uses in Section 9.11.

9.6.2 Reading a file into collections

Collections become useful the moment data arrives from outside. **Listing 9.13** reads a small text file twice — once word by word with a `Scanner`, once line by line with a `BufferedReader` — counts the words with a `TreeMap`, and reports the three most frequent words. The file `words.txt` sits next to the program:

```
Wash the towels and fold the towels
Iron the shirts
Wash the shirts and the sheets
Fold the sheets
```

Listing 9.13 `WordFile.java`: a file read into a `List` of words and a `List` of lines, the words counted in a `TreeMap`, and the most frequent found by sorting the entries.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Scanner;
import java.util.TreeMap;

public class WordFile {
    public static void main(String[] args) {
        String fileName = args.length > 0 ? args[0] : "words.txt";

        // 1st method: a Scanner hands us one word at a time.
        List<String> words = new ArrayList<>();
        try (Scanner fileScan = new Scanner(new File(fileName))) {
            while (fileScan.hasNext()) {
                words.add(fileScan.next().toLowerCase());
            }
        } catch (IOException e) {
            System.err.println(e);
            return;
        }
        System.out.println(words.size() + " words: " + words);

        // 2nd method: a BufferedReader hands us one line at a time.
        List<String> stringList = new ArrayList<>();
        try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
            String currentLine;
            while ((currentLine = br.readLine()) != null) {
                stringList.add(currentLine);
            }
        } catch (IOException e) {
            System.err.println(e);
            return;
        }
        System.out.println("Now we want to print the list out:");
        for (int i = 0; i < stringList.size(); i++) {
            System.out.println((i + 1) + ": " + stringList.get(i));
        }
```

Listing 9.13 (Continued) *WordFile.java: a file read into a List of words and a List of lines, the words counted in a TreeMap, and the most frequent found by sorting the entries.*

```
        // Count every word; a TreeMap keeps the words in alphabetical order.
        Map<String, Integer> counts = new TreeMap<>();
        for (String w : words) {
            counts.put(w, counts.getOrDefault(w, 0) + 1);
        }
        System.out.println(counts.size() + " distinct: " + counts);

        // The three most frequent words: most first, ties alphabetically.
        List<Map.Entry<String, Integer>> entries = new ArrayList<>(counts.entrySet());
        entries.sort((a, b) -> {
            int byCount = Integer.compare(b.getValue(), a.getValue());   // larger first
            return byCount != 0 ? byCount : a.getKey().compareTo(b.getKey());
        });
        System.out.println("top 3: " + entries.subList(0, 3));
    }
}
```

```
19 words: [wash, the, towels, and, fold, the, towels, iron, the, shirts, wash, the, shirts,
and, the, sheets, fold, the, sheets]
Now we want to print the list out:
1: Wash the towels and fold the towels
2: Iron the shirts
3: Wash the shirts and the sheets
4: Fold the sheets
8 distinct: {and=2, fold=2, iron=1, sheets=2, shirts=2, the=6, towels=2, wash=2}
top 3: [the=6, and=2, fold=2]
```

Both readers come from Chapter 5 and are opened in `try`-with-resources. The `Scanner`'s `next()` returns one word at a time, lowercased so that "Wash" and "wash" count as one; the `BufferedReader`'s `readLine()` returns one line at a time and `null` at the end. (Code that closes a `Scanner` in `finally` with `fileScan.close()` has a hole: if the file is missing, `fileScan` is still `null`, and the `finally` block throws a `NullPointerException` that hides the real error.) `java WordFile nothere.txt` prints `java.io.FileNotFoundException: nothere.txt (No such file or directory)`.

The counting line `counts.put(w, counts.getOrDefault(w, 0) + 1)` is the long form of `merge`. The `TreeMap` then prints the eight distinct words alphabetically. To find the most frequent words we copy the entries into a list and sort it with a comparator written as a lambda: compare the counts in *reverse* (`Integer.compare(b.getValue(), a.getValue())`, larger first), and, only when the counts are equal, the words alphabetically. `subList(0, 3)` is a view of the first three. The tie among the six words that occur twice is broken alphabetically, so `and` and `fold` win.

Listing 9.14 is shorter, as file handling usually is in Python.

Listing 9.14 `word_file.py`: the same report in Python, with a sort key that orders by count descending and word ascending, and `Counter.most_common`.

```python
"""word_file.py: read a file into a list, count its words, report the most frequent."""
import sys
from collections import Counter


def main():
    file_name = sys.argv[1] if len(sys.argv) > 1 else "words.txt"
    try:
        with open(file_name, encoding="utf-8") as f:
            text = f.read()
    except OSError as e:
        print(e, file=sys.stderr)
        return

    words = [w.lower() for w in text.split()]        # split() cuts at any whitespace
    print(len(words), "words:", words)

    string_list = text.splitlines()                  # one string per line
    print("Now we want to print the list out:")
    for i, line in enumerate(string_list, start=1):
        print(f"{i}: {line}")

    counts: dict[str, int] = {}
    for w in words:
        counts[w] = counts.get(w, 0) + 1
    print(len(counts), "distinct:", dict(sorted(counts.items())))

    # The three most frequent words: most first, ties alphabetically.
    ranked = sorted(counts.items(), key=lambda kv: (-kv[1], kv[0]))
    print("top 3:", ranked[:3])
    print("Counter:", Counter(words).most_common(3))


if __name__ == "__main__":
    main()
```

```
19 words: ['wash', 'the', 'towels', 'and', 'fold', 'the', 'towels', 'iron', 'the',
'shirts', 'wash', 'the', 'shirts', 'and', 'the', 'sheets', 'fold', 'the', 'sheets']
Now we want to print the list out:
1: Wash the towels and fold the towels
2: Iron the shirts
3: Wash the shirts and the sheets
4: Fold the sheets
8 distinct: {'and': 2, 'fold': 2, 'iron': 1, 'sheets': 2, 'shirts': 2, 'the': 6, 'towels':
2, 'wash': 2}
top 3: [('the', 6), ('and', 2), ('fold', 2)]
Counter: [('the', 6), ('wash', 2), ('towels', 2)]
```

`text.split()` with no argument splits at any run of whitespace, including line breaks, and `text.splitlines()` gives the lines without their line-break characters. The sort key `lambda kv: (-kv[1], kv[0])` returns a *tuple* for each pair: Python compares tuples element by element, so pairs are ordered by the negated count (larger counts first) and then by the word. Negating a number is the Python idiom for "this part descending". The last line shows that `Counter.most_common(3)` also finds `the` first, but breaks the tie among the twos by *first appearance* — `wash`, then `towels` — rather than alphabetically. Both answers are correct; they answer different questions, and a report that promises "ties alphabetically" must sort explicitly.

9.7 Queues, Stacks, Deques, and Priority Queues

Some collections are defined less by how they store elements than by *which element comes out next*. **Figure 9.7** shows the three disciplines.

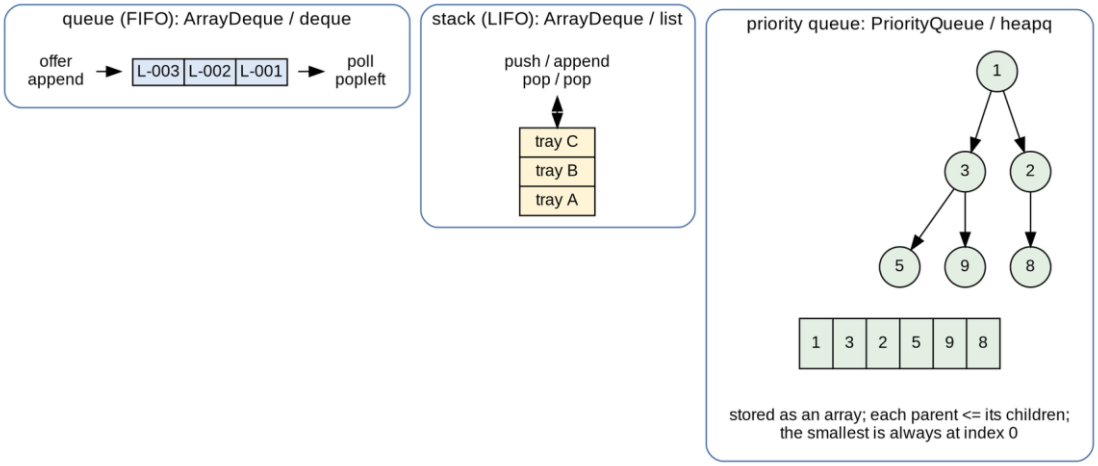


Figure 9.7 Three disciplines for taking elements out. A queue releases the oldest element (FIFO), a stack the newest (LIFO), and a priority queue the smallest, which a binary heap keeps at index 0 of its array.

- A **queue** is first in, first out (FIFO), like the line at the laundry counter: `offer` adds at the back, `poll` removes from the front, `peek` looks at the front without removing.
- A **stack** is last in, first out (LIFO), like a pile of trays: `push` puts on top, `pop` takes from the top.
- A **priority queue** releases the *smallest* element first, whatever the order of arrival — the smallest by natural order or by a `Comparator` that you supply.

Java's `Deque` interface (pronounced "deck", a *double-ended queue*) allows insertion and removal at both ends, so one class, `ArrayDeque`, serves as both queue and stack. It is a circular array: the front and back move around a fixed block of slots, so both ends cost $O(1)$ without shifting anything. **Listing 9.15** uses all three disciplines.

Listing 9.15 `QueueDemo.java`: an `ArrayDeque` as a FIFO queue, as a LIFO stack and as a deque; a `PriorityQueue` with natural order and with `Comparator.reverseOrder()`.

```
import java.util.ArrayDeque;
import java.util.Comparator;
import java.util.Deque;
import java.util.PriorityQueue;
import java.util.Queue;

public class QueueDemo {
    public static void main(String[] args) {
        // A queue: first in, first out (FIFO).
        Queue<String> washQueue = new ArrayDeque<>();
        washQueue.offer("L-001");
        washQueue.offer("L-002");
        washQueue.offer("L-003");
        System.out.println("queue " + washQueue + ", peek " + washQueue.peek());
        System.out.println("poll " + washQueue.poll() + ", poll " + washQueue.poll()
                + ", left " + washQueue);

        // A stack: last in, first out (LIFO). Deque = double-ended queue.
        Deque<String> trays = new ArrayDeque<>();
        trays.push("tray A");
        trays.push("tray B");
        trays.push("tray C");
        System.out.println("stack " + trays + ", pop " + trays.pop() + ", pop " +
trays.pop());

        // Both ends of one deque.
        Deque<Integer> d = new ArrayDeque<>();
        d.addLast(2);
        d.addLast(3);
        d.addFirst(1);
        System.out.println("deque " + d + ", pollFirst " + d.pollFirst()
                + ", pollLast " + d.pollLast());

        // A priority queue: the smallest element leaves first.
        Queue<Integer> pq = new PriorityQueue<>();
        for (int x : new int[] {5, 1, 8, 3, 9, 2}) {
            pq.offer(x);
        }
        System.out.println("heap array " + pq);          // NOT sorted: a heap
        StringBuilder out = new StringBuilder();
        while (!pq.isEmpty()) {
            out.append(pq.poll()).append(' ');
        }
        System.out.println("poll order " + out.toString().trim());
```

Listing 9.15 (Continued) *QueueDemo.java: an ArrayDeque as a FIFO queue, as a LIFO stack and as a deque; a PriorityQueue with natural order and with Comparator.reverseOrder().*

```
        // Largest first, with a Comparator.
        Queue<Integer> maxFirst = new PriorityQueue<>(Comparator.reverseOrder());
        for (int x : new int[] {5, 1, 8, 3, 9, 2}) {
            maxFirst.offer(x);
        }
        System.out.println("max first: " + maxFirst.poll() + " " + maxFirst.poll()
                + " " + maxFirst.poll());
        System.out.println("empty poll -> " + new ArrayDeque<String>().poll());
    }
}
```

```
queue [L-001, L-002, L-003], peek L-001
poll L-001, poll L-002, left [L-003]
stack [tray C, tray B, tray A], pop tray C, pop tray B
deque [1, 2, 3], pollFirst 1, pollLast 3
heap array [1, 3, 2, 5, 9, 8]
poll order 1 2 3 5 8 9
max first: 9 8 5
empty poll -> null
```

The queue releases `L-001` first, then `L-002`: the order of arrival. The stack prints `[tray C, tray B, tray A]` — an `ArrayDeque` used as a stack prints from the top down — and pops `tray C` first. The deque accepts `addFirst` and `addLast` and gives `pollFirst` and `pollLast`. Each `Queue` operation comes in two flavors: `offer`, `poll` and `peek` return `false` or `null` when they cannot do their job, as the last line shows, while `add`, `remove` and `element` throw an exception instead.

The priority queue deserves a slow look. After six `offer`s, it prints `[1, 3, 2, 5, 9, 8]`: *not* sorted. A `PriorityQueue` is a **binary heap**, an array read as a complete binary tree in which the children of index i sit at indices $2i + 1$ and $2i + 2$, and every parent is no larger than its children (**Figure 9.7**, right). That rule puts the smallest element at index 0 but says nothing about the rest. `poll` removes index 0, moves the last element to the top and lets it sink to its place, which restores the rule in at most one step per level. A heap of n elements has $\lfloor \log_2 n \rfloor + 1$ levels, so `offer` and `poll` cost $O(\log n)$ and `peek` costs $O(1)$ [9]. Polling until empty yields `1 2 3 5 8 9`: sorted, one element at a time. To take the *largest* first, give the constructor `Comparator.reverseOrder()`.

Listing 9.16 shows Python's tools: `collections.deque` for queues and deques, a plain `list` for stacks, and the module `heapq`, which applies the heap rules to an ordinary list.

Listing 9.16 `queue_demo.py`: `deque` as a queue and a deque, `list` as a stack, and `heapq` as a priority queue.

```
"""queue_demo.py: deque as queue and stack, heapq as a priority queue."""
import heapq
from collections import deque


def main():
    # A queue: first in, first out (FIFO).
    wash_queue = deque()
    wash_queue.append("L-001")
    wash_queue.append("L-002")
    wash_queue.append("L-003")
    print(f"queue {list(wash_queue)}, peek {wash_queue[0]}")
    first, second = wash_queue.popleft(), wash_queue.popleft()
    print(f"poll {first}, poll {second}, left {list(wash_queue)}")

    # A stack: last in, first out (LIFO). A plain list is the usual stack.
    trays = []
    trays.append("tray A")
    trays.append("tray B")
    trays.append("tray C")
    # An f-string is built left to right, so the list is shown before the pops.
    print(f"stack {trays[::-1]}, pop {trays.pop()}, pop {trays.pop()}")

    # Both ends of one deque.
    d = deque()
    d.append(2)
    d.append(3)
    d.appendleft(1)
    print(f"deque {list(d)}, pollFirst {d.popleft()}, pollLast {d.pop()}")

    # A priority queue: the smallest element leaves first.
    pq = []
    for x in [5, 1, 8, 3, 9, 2]:
        heapq.heappush(pq, x)
    print("heap array", pq)                              # NOT sorted: a heap
    print("poll order", " ".join(str(heapq.heappop(pq)) for _ in range(len(pq))))
```

Listing 9.16 (Continued) *queue_demo.py: deque as a queue and a deque, list as a stack, and heapq as a priority queue.*

```
    # Largest first: push the negatives.
    max_first = []
    for x in [5, 1, 8, 3, 9, 2]:
        heapq.heappush(max_first, -x)
    print("max first:", -heapq.heappop(max_first), -heapq.heappop(max_first),
          -heapq.heappop(max_first))
    try:
        deque().popleft()
    except IndexError as e:
        print("empty popleft -> IndexError:", e)


if __name__ == "__main__":
    main()
```

```
queue ['L-001', 'L-002', 'L-003'], peek L-001
poll L-001, poll L-002, left ['L-003']
stack ['tray C', 'tray B', 'tray A'], pop tray C, pop tray B
deque [1, 2, 3], pollFirst 1, pollLast 3
heap array [1, 3, 2, 5, 9, 8]
poll order 1 2 3 5 8 9
max first: 9 8 5
empty popleft -> IndexError: pop from an empty deque
```

`deque.popleft()` is the $O(1)$ removal from the front; the same operation on a `list`, `pop(0)`, shifts every remaining element and costs $O(n)$. That is the single most common performance mistake in Python programs that process queues. The heap array is identical to Java's, `[1, 3, 2, 5, 9, 8]`, because both libraries implement the same textbook algorithm. `heapq` has no reverse option; the idiom for "largest first" is to push negated numbers and negate them again when they come out. An empty `deque` raises `IndexError` rather than returning `None`. (The `print` of the stack uses an f-string, whose parts are evaluated left to right, and shows the list reversed so that the top comes first as in Java. With `print(trays, trays.pop(), ...)`, the arguments would all be evaluated *before* printing, and the list would be shown after both pops.)

9.8 Generics from Zero

We have written `List<String>`, `Map<Integer, String>` and `PriorityQueue<Integer>` dozens of times. It is time to understand the angle brackets.

> **DEFINITION.** A **generic class** (or generic interface) is a class written with one or more **type parameters**, placeholder names for types, written in angle brackets after the class name: `class GList<T>`. A use of the class supplies a **type argument** for each parameter, `GList<Integer>`, and the compiler checks every use as if `T` had been replaced by `Integer` throughout. A **generic method** declares its own type parameters before its return type: `static <T> T lastOf(List<T> list)`.

Put differently, generics let us write a class that does not care what kind of object it manipulates, as long as they are all of the same kind, and let the compiler hold every user of the class to the kind they chose.

9.8.1 Life before generics

Generics arrived in Java 5, in 2004 [11]. Before that, a `List` held `Object`s, and **Listing 9.17** shows what that meant.

Listing 9.17 `RawTypes.java`: a raw `List` accepts an `Integer` among the names and fails only when it is cast; the generic `List<String>` needs no cast and would reject the `Integer` at compile time.

```
import java.util.ArrayList;
import java.util.List;

public class RawTypes {
    @SuppressWarnings({"rawtypes", "unchecked"})
    static void beforeGenerics() {
        List names = new ArrayList();      // a raw List: it holds Objects
        names.add("Siti");
        names.add("Budi");
        names.add(42);                     // accepted without a murmur
        for (int i = 0; i < names.size(); i++) {
            try {
                String name = (String) names.get(i);   // the cast is our only check
                System.out.println(name.toUpperCase());
            } catch (ClassCastException e) {
                System.out.println("index " + i + ": ClassCastException");
            }
        }
    }

    static void withGenerics() {
        List<String> names = new ArrayList<>();
        names.add("Siti");
        names.add("Budi");
        // names.add(42);                  // compile error: int cannot be converted to
String
        for (String name : names) {        // no cast needed
            System.out.println(name.toUpperCase());
        }
    }

    public static void main(String[] args) {
        System.out.println("-- before generics --");
        beforeGenerics();
        System.out.println("-- with generics --");
        withGenerics();
    }
}
```

```
-- before generics --
SITI
BUDI
index 2: ClassCastException
-- with generics --
SITI
BUDI
```

In `beforeGenerics`, the raw list accepts `"Siti"`, `"Budi"` and the number `42` without a word, because all three are `Object`s. The damage appears only when the elements come out: each `get(i)` returns an `Object`, the cast `(String)` is the programmer's claim that it is really a string, and for index 2 the claim is false, so the virtual machine throws `ClassCastException`. In a real program the `add(42)` and the failing cast may be in different files, written months apart, and the stack trace points at the cast, not at the mistake.

In `withGenerics`, the declaration `List<String>` makes the compiler do the checking. The for-each variable can be a `String` directly, with no cast, and the commented line `names.add(42)`, if you remove its `//`, is refused before the program can run:

```
RawTypes.java:25: error: incompatible types: int cannot be converted to String
        names.add(42);
                  ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
1 error
```

That is the whole purpose of generics: to move a family of errors from run time, where a user meets them, to compile time, where the programmer does.

Python's lists have always held anything. **Listing 9.18** shows that the same mistake runs, and how type hints let mypy find it.

Listing 9.18 `raw_types.py`: a Python list accepts anything; the hint `list[str]` lets mypy reject the `append(42)` before the program runs.

```
"""raw_types.py: a Python list accepts anything; type hints let mypy check it."""


def without_hints():
    names = []                          # any object may go in
    names.append("Siti")
    names.append("Budi")
    names.append(42)                    # accepted without a murmur
    for i, name in enumerate(names):
        try:
            print(name.upper())
        except AttributeError:
            print(f"index {i}: AttributeError")


def with_hints() -> None:
    names: list[str] = []               # a promise for mypy: only str goes in
    names.append("Siti")
    names.append("Budi")
    names.append(42)                    # mypy reports this line; Python runs it
    for name in names:
        print(name.upper() if isinstance(name, str) else f"{name!r} slipped in")


def main():
    print("-- without hints --")
    without_hints()
    print("-- with hints --")
    with_hints()


if __name__ == "__main__":
    main()
```

```
-- without hints --
SITI
BUDI
index 2: AttributeError
-- with hints --
SITI
BUDI
42 slipped in
```

Both functions run, and in both the integer slips in. mypy says nothing about `without_hints`, whose list has no declared element type, and one precise thing about `with_hints`:

```
raw_types.py:20: error: Argument 1 to "append" of "list" has incompatible type "int";
expected "str"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

Java's generics are enforced by the compiler every program must pass; Python's hints by a separate tool that you choose to run.

9.8.2 A generic class: GList

Chapter 8 built an immutable list of *integers* from `nil` and `cons`. A list of strings would need a second class, identical except for the word `int`, and a list of orders a third. **Listing 9.19** writes the list once, for any element type `T`.

Listing 9.19 `GListDemo.java`: `GList<T>`, a generic immutable list with a recursive `toString` helper, used with `Integer` and with `String`.

```
import java.util.ArrayList;
import java.util.List;

public class GListDemo {
    public static void main(String[] args) {
        GList<Integer> myGList = new GList<>();
        System.out.printf("myGList: %s%n", myGList);
        int head = 10;
        GList<Integer> myGList2 = new GList<>(head, myGList);
        System.out.printf("myGList2: %s%n", myGList2);
        head = 50;
        GList<Integer> myGList3 = new GList<>(head, myGList2);
        System.out.printf("myGList3: %s%n", myGList3);

        // The same class, a different type argument.
        GList<String> towels = new GList<>("white", new GList<>("blue", GList.nil()));
        String first = towels.head();                  // no cast: head() returns String
        System.out.println("towels: " + towels + ", head " + first
                + ", tail " + towels.tail() + ", size " + towels.size());
        try {
            GList.<String>nil().tail();
        } catch (IllegalStateException e) {
            System.out.println("IllegalStateException: " + e.getMessage());
        }
    }
}

// A general (generic) immutable list: T stands for any class type.
class GList<T> {
    private final boolean empty;
    private final T head;
    private final GList<T> tail;

    public GList() {                                   // the empty list
        empty = true;
        head = null;
        tail = null;
    }

    public GList(T head, GList<T> tail) {              // a non-empty list
        this.empty = false;
        this.head = head;
        this.tail = tail;
    }
```

Listing 9.19 (Continued) *GListDemo.java: GList<T>, a generic immutable list with a recursive toString helper, used with Integer and with String.*

```
    public static <T> GList<T> nil() {                // a generic method
        return new GList<>();
    }

    public boolean empty() {
        return empty;
    }

    public T head() {
        if (empty) {
            throw new IllegalStateException("Trying to access head of an empty list");
        }
        return head;
    }

    public GList<T> tail() {
        if (empty) {
            throw new IllegalStateException("Trying to access tail of an empty list");
        }
        return tail;
    }

    public int size() {                               // recursive: 1 + size of the tail
        return empty ? 0 : 1 + tail.size();
    }

    @Override
    public String toString() {
        return toStringHelper(this, new ArrayList<T>());
    }

    // A recursive helper with an extra parameter, the accumulator.
    private String toStringHelper(GList<T> list, List<T> accumulator) {
        if (list == null) {
            throw new IllegalStateException("next element of list was null. "
                    + "Should either be another list or nil. List till now: " +
accumulator);
        } else if (list.empty()) {
            return accumulator.toString();
        } else {
            accumulator.add(list.head());
            return toStringHelper(list.tail(), accumulator);
        }
    }
}
```

```
myGList: []
myGList2: [10]
myGList3: [50, 10]
towels: [white, blue], head white, tail [blue], size 2
IllegalStateException: Trying to access tail of an empty list
```

In `class GList<T>`, the type parameter `T` is used wherever a type is expected: `head` has type `T`, `tail` has type `GList<T>`, `head()` returns a `T`. By convention type parameters are single capitals — `T` for type, `E` for element, `K` and `V` for key and value — so that they stand out from class names.

In `main`, `GList<Integer>` supplies the type argument `Integer`; each `new GList<>(head, ...)` puts a boxed `int` in front, giving `[]`, `[10]` and `[50, 10]`. The *same class* then holds `String`s, and `towels.head()` returns a `String` without any cast.

The `toString` method is the recursion-with-helper pattern of Section 9.2.3. The public method has the signature every object needs, `toString()`, and hands the work to the private

`toStringHelper(list, accumulator)`, whose extra parameter is an `ArrayList<T>` that collects the elements on the way down. The base case is the empty list, which returns the accumulator's own `toString`, `[50, 10]`. The helper also guards against a broken chain (a `null` tail) with a message showing what has been collected so far — useful when debugging a hand-built list. `size()` is recursion without a helper: one plus the size of the tail.

The static method `nil()` is our first *generic method*. A static method cannot use the class's `T` (static members belong to the class, not to any particular `GList<Integer>` or `GList<String>`), so it declares its own: `public static <T> GList<T> nil()`. The compiler infers `T` from the context, so `new GList<>("blue", GList.nil())` gets a `GList<String>`. Where there is no context, the type argument can be given explicitly: `GList.<String>nil().tail()`.

> ⚠ **WARNING.** Older code sometimes declares such methods with a *raw* return type, `public static GList nil()`, and writes `GList tail()` without `<T>`. It compiles, with warnings, and quietly switches off checking for every caller: a raw `GList` can be assigned to a `GList<String>` whatever it really contains. Compile with `javac -Xlint:all`, and treat every "unchecked" or "rawtypes" warning as a bug to fix. Every listing in this chapter compiles with no warnings except where raw types are shown on purpose, and there the warnings are suppressed explicitly.

Listing 9.20 is the Python version.

Listing 9.20 `glist_demo.py`: the generic list with `TypeVar` and `Generic[T]`, checked by mypy.

```python
"""glist_demo.py: a generic immutable list with typing.Generic."""
from __future__ import annotations

from typing import Generic, TypeVar

T = TypeVar("T")                        # T stands for any type


class GList(Generic[T]):
    _head: T                            # declared here, set only in non-empty lists
    _tail: GList[T]

    def __init__(self) -> None:         # the empty list
        self._empty = True

    @staticmethod
    def nil() -> GList[T]:
        return GList()

    @staticmethod
    def cons(head: T, tail: GList[T]) -> GList[T]:
        lst: GList[T] = GList()         # a non-empty list
        lst._empty = False
        lst._head = head
        lst._tail = tail
        return lst

    def empty(self) -> bool:
        return self._empty

    def head(self) -> T:
        if self._empty:
            raise ValueError("Trying to access head of an empty list")
        return self._head

    def tail(self) -> GList[T]:
        if self._empty:
            raise ValueError("Trying to access tail of an empty list")
        return self._tail

    def size(self) -> int:              # recursive: 1 + size of the tail
        return 0 if self._empty else 1 + self.tail().size()

    def __str__(self) -> str:
        return self._str_helper(self, [])
```

Listing 9.20 (Continued) *glist_demo.py: the generic list with TypeVar and Generic[T], checked by mypy.*

```
    def _str_helper(self, lst: GList[T], accumulator: list[T]) -> str:
        """A recursive helper with an extra parameter, the accumulator."""
        if lst.empty():
            return "[" + ", ".join(str(x) for x in accumulator) + "]"
        accumulator.append(lst.head())
        return self._str_helper(lst.tail(), accumulator)


def main() -> None:
    my_glist: GList[int] = GList()
    print(f"myGList: {my_glist}")
    head = 10
    my_glist2 = GList.cons(head, my_glist)
    print(f"myGList2: {my_glist2}")
    head = 50
    my_glist3 = GList.cons(head, my_glist2)
    print(f"myGList3: {my_glist3}")

    towels: GList[str] = GList.cons("white", GList.cons("blue", GList.nil()))
    first: str = towels.head()
    print(f"towels: {towels}, head {first}, tail {towels.tail()}, size {towels.size()}")
    try:
        GList.nil().tail()
    except ValueError as e:
        print("ValueError:", e)


if __name__ == "__main__":
    main()
```

```
myGList: []
myGList2: [10]
myGList3: [50, 10]
towels: [white, blue], head white, tail [blue], size 2
ValueError: Trying to access tail of an empty list
```

Python spells the type parameter in two steps. `T = TypeVar("T")` creates a type variable, and `class GList(Generic[T])` declares that `GList` is parameterized by it. After that, `GList[int]` and `GList[str]` are types that mypy understands, and `towels.head()` is known to be a `str`. (Python 3.12 added a shorter syntax, `class GList[T]:`, with no `TypeVar` line [12]; the listing uses the older one so that it runs on Python 3.10 and 3.11 as well.) Two further details: `from __future__ import annotations` lets a method mention `GList[T]` inside the class that is still being defined, and the class-level annotations `_head: T` and `_tail: GList[T]` declare attributes that only non-empty lists receive, in `cons`. Python has no overloaded constructors, so `nil()` and `cons()` are the two ways to build a list, just as Chapter 8's Java `List` had them.

9.8.3 Generic methods, and a Pair at last

Section 9.2.3 ended with a wish: a class that holds two values of any two types, so that `fibonacciPair` could return something that says what it means. **Listing 9.21** grants it, together with a few generic methods.

Listing 9.21 `GenericMethods.java`: a generic class with two type parameters, `Pair<A, B>`, and four generic methods, among them `removeDuplicatesFromSortedList`.

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class GenericMethods {
    // <A, B> before the return type declares the method's own type parameters.
    static <A, B> Pair<B, A> swap(Pair<A, B> p) {
        return new Pair<>(p.getSecond(), p.getFirst());
    }

    static <T> T lastOf(List<T> list) {
        return list.get(list.size() - 1);
    }

    // {F(n), F(n-1)} again, now as a Pair instead of a long[].
    static Pair<Long, Long> fibonacciPair(int n) {
        if (n <= 1) {
            return new Pair<>((long) n, 0L);
        }
        Pair<Long, Long> tmp = fibonacciPair(n - 1);
        return new Pair<>(tmp.getFirst() + tmp.getSecond(), tmp.getFirst());
    }

    // Keeps the first of every run of equal neighbors. Correct for SORTED input.
    static <T> List<T> removeDuplicatesFromSortedList(List<T> input) {
        List<T> result = new ArrayList<>();
        T previous = null;
        Iterator<T> iter = input.iterator();
        while (iter.hasNext()) {
            T current = iter.next();
            if (result.isEmpty() || !previous.equals(current)) {
                result.add(current);
            }
            previous = current;
        }
        return result;
    }

    public static void main(String[] args) {
        Pair<String, Integer> order = new Pair<>("L-001", 24500);
        Pair<Integer, String> swapped = swap(order);          // types inferred
        System.out.println(order + " swapped is " + swapped);

        int price = order.getSecond();                        // unboxed, no cast
        System.out.println("price + 500 = " + (price + 500));
```

Listing 9.21 (Continued) *GenericMethods.java: a generic class with two type parameters, Pair<A, B>, and four generic methods, among them removeDuplicatesFromSortedList.*

```
        System.out.println("lastOf " + lastOf(List.of("wash", "dry", "iron")));
        System.out.println("lastOf " + GenericMethods.<Integer>lastOf(List.of(3, 1, 4)));

        System.out.println("fibonacciPair(10) = " + fibonacciPair(10));
        System.out.println("fibonacciPair(90).getFirst() = " +
fibonacciPair(90).getFirst());

        System.out.println(removeDuplicatesFromSortedList(
                List.of("cat", "cat", "dog", "monkey", "monkey")));
        System.out.println(removeDuplicatesFromSortedList(List.of(1, 1, 2, 3, 3, 3)));
        System.out.println(removeDuplicatesFromSortedList(new ArrayList<Double>()));
        System.out.println(removeDuplicatesFromSortedList(List.of(3, 1, 3)));   //
unsorted!
    }
}

// A generic class with two type parameters.
final class Pair<A, B> {
    private final A first;
    private final B second;

    Pair(A first, B second) {
        this.first = first;
        this.second = second;
    }

    A getFirst() {
        return first;
    }

    B getSecond() {
        return second;
    }

    @Override
    public String toString() {
        return "(" + first + ", " + second + ")";
    }
}
```

```
(L-001, 24500) swapped is (24500, L-001)
price + 500 = 25000
lastOf iron
lastOf 4
fibonacciPair(10) = (55, 34)
fibonacciPair(90).getFirst() = 2880067194370816120
[cat, dog, monkey]
[1, 2, 3]
[]
[3, 1, 3]
```

`final class Pair<A, B>` has two type parameters, one per component. `new Pair<>("L-001", 24500)` is a `Pair<String, Integer>`, and `order.getSecond()` is an `Integer` that unboxes into the `int` variable `price` without a cast.

A generic method writes its type parameters between the modifiers and the return type. `static <A, B> Pair<B, A> swap(Pair<A, B> p)` reads: “for any types `A` and `B`, given a pair of an `A` and a `B`, return a pair of a `B` and an `A`”. The caller never mentions `A` or `B`; the compiler **infers** them from the argument, so `swap(order)` has type `Pair<Integer, String>`. `lastOf` works for a list of any type and returns that type; `GenericMethods.<Integer>lastOf(...)` shows the rarely needed explicit form. `fibonacciPair` now returns a `Pair<Long, Long>` — the same algorithm as in

Listing 9.3, but the result prints as `(55, 34)` and its parts are called `getFirst()` and `getSecond()` instead of `[0]` and `[1]`.

`removeDuplicatesFromSortedList` is a generic method that works for a list of *anything* whose `equals` is meaningful. It keeps the first element of every run of equal neighbors. On a sorted list, equal elements are neighbors, so this removes all duplicates in one pass, $O(n)$: `[cat, cat, dog, monkey, monkey]` becomes `[cat, dog, monkey]`. The last call shows why the name says *sorted*: on `[3, 1, 3]` the two 3s are not neighbors, and nothing is removed. For unsorted input the right tool is a set — `new ArrayList<>(new LinkedHashSet<>(list))` keeps first occurrences in order, also in $O(n)$ expected time — or sorting first, in $O(n \log n)$. You may meet an earlier version of this method that starts with `result.add(input.get(0))` and tests `previous != null`: it throws `IndexOutOfBoundsException` on an empty list, and it misbehaves when the list contains `null`. The version in the listing tests `result.isEmpty()` instead and handles both; the call with an empty `ArrayList<Double>` proves the first point.

Listing 9.22 is the Python counterpart.

Listing 9.22 `generic_methods.py`: `Pair(Generic[A, B])` and generic functions typed with `TypeVar`.

```
"""generic_methods.py: a generic Pair class and generic functions with TypeVar."""
from __future__ import annotations

from typing import Generic, Sequence, TypeVar

A = TypeVar("A")
B = TypeVar("B")
T = TypeVar("T")


class Pair(Generic[A, B]):
    """A generic class with two type parameters."""

    def __init__(self, first: A, second: B) -> None:
        self._first = first
        self._second = second

    @property
    def first(self) -> A:
        return self._first

    @property
    def second(self) -> B:
        return self._second

    def __str__(self) -> str:
        return f"({self._first}, {self._second})"


def swap(p: Pair[A, B]) -> Pair[B, A]:
    return Pair(p.second, p.first)


def last_of(seq: Sequence[T]) -> T:
    return seq[-1]


def fibonacci_pair(n: int) -> Pair[int, int]:
    """(F(n), F(n-1)) again, now as a Pair instead of a bare tuple."""
    if n <= 1:
        return Pair(n, 0)
    tmp = fibonacci_pair(n - 1)
    return Pair(tmp.first + tmp.second, tmp.first)
```

Listing 9.22 (Continued) *generic_methods.py: Pair(Generic[A, B]) and generic functions typed with TypeVar.*

```
def remove_duplicates_from_sorted_list(items: Sequence[T]) -> list[T]:
    """Keeps the first of every run of equal neighbors. Correct for SORTED input."""
    result: list[T] = []
    for current in items:
        if not result or result[-1] != current:
            result.append(current)
    return result


def main() -> None:
    order: Pair[str, int] = Pair("L-001", 24500)
    swapped = swap(order)                          # mypy infers Pair[int, str]
    print(f"{order} swapped is {swapped}")

    price: int = order.second
    print("price + 500 =", price + 500)

    print("lastOf", last_of(["wash", "dry", "iron"]))
    print("lastOf", last_of((3, 1, 4)))

    print("fibonacciPair(10) =", fibonacci_pair(10))
    print("fibonacciPair(90).getFirst() =", fibonacci_pair(90).first)

    print(remove_duplicates_from_sorted_list(["cat", "cat", "dog", "monkey", "monkey"]))
    print(remove_duplicates_from_sorted_list([1, 1, 2, 3, 3, 3]))
    print(remove_duplicates_from_sorted_list([]))
    print(remove_duplicates_from_sorted_list([3, 1, 3]))     # unsorted!


if __name__ == "__main__":
    main()
```

```
(L-001, 24500) swapped is (24500, L-001)
price + 500 = 25000
lastOf iron
lastOf 4
fibonacciPair(10) = (55, 34)
fibonacciPair(90).getFirst() = 2880067194370816120
['cat', 'dog', 'monkey']
[1, 2, 3]
[]
[3, 1, 3]
```

A generic function in Python simply uses a `TypeVar` in its signature: `def swap(p: Pair[A, B]) -> Pair[B, A]` says the same thing as the Java header, and mypy infers `A` and `B` at each call. `last_of(seq: Sequence[T]) -> T` accepts a list, a tuple, a string or a `range`, because all are `Sequence`s. (For two values that need no names, a plain tuple is the Python habit; a `Pair` class earns its keep when the components deserve names, or when the type should appear in signatures.)

9.8.4 Bounded type parameters

Try to add a method to `GList<T>` that checks whether the list is sorted. It must compare neighbors, `a.compareTo(b)`, and the compiler refuses: `T` could be *any* class, and not every class has a `compareTo`. Java classes have no natural ordering in general — what would it mean for one `Scanner` to come before another? We need to say "any type `T`, *provided* it can be compared with other `T`s". That is a **bounded type parameter**:

```
class GList2<T extends Comparable<T>> { ... }
```

Read `T extends Comparable<T>` as "`T` must be a type that implements `Comparable<T>`". (In a bound, `extends` covers both extending a class and implementing an interface.) Inside the class, the compiler now knows that every `T` has a method `compareTo(T)`. Outside, only such types are accepted as type arguments: `Integer`, `String`, `Double` and your own `Comparable` classes, but not `Object`. **Listing 9.23** is the result.

Listing 9.23 `GList2Demo.java`: a list whose element type is bounded by `Comparable`, with a recursive `sorted` method, and a bounded generic method `max`.

```
import java.util.List;

public class GList2Demo {
    // A bounded generic method: T must know how to compare itself with another T.
    static <T extends Comparable<T>> T max(List<T> items) {
        T best = items.get(0);
        for (T x : items) {
            if (x.compareTo(best) > 0) {
                best = x;
            }
        }
        return best;
    }

    public static void main(String[] args) {
        GList2<Integer> myGList4 = new GList2<>();
        System.out.printf("myGList4: %s%n", myGList4);
        System.out.printf("myGList4 is sorted: %b%n", myGList4.sorted(myGList4));
        int head = 5;
        GList2<Integer> myGList5 = new GList2<>(head, myGList4);
        System.out.printf("myGList5: %s%n", myGList5);
        System.out.printf("myGList5 is sorted: %b%n", myGList5.sorted(myGList5));
        head = 10;
        GList2<Integer> myGList6 = new GList2<>(head, myGList5);
        System.out.printf("myGList6: %s%n", myGList6);
        System.out.printf("myGList6 is sorted: %b%n", myGList6.sorted(myGList6));
        head = 3;
        GList2<Integer> myGList7 = new GList2<>(head, myGList5);
        System.out.printf("myGList7: %s%n", myGList7);
        System.out.printf("myGList7 is sorted: %b%n", myGList7.sorted(myGList7));

        GList2<String> words = new GList2<>("iron", new GList2<>("wash", new GList2<>()));
        System.out.printf("words: %s is sorted: %b%n", words, words.sorted());

        System.out.println("max " + max(List.of(3, 9, 4)) + ", max "
                + max(List.of("wash", "dry", "iron")));
    }
}

// T is bounded: only types that implement Comparable<T> are allowed.
class GList2<T extends Comparable<T>> {
    private final boolean empty;
    private final T head;
    private final GList2<T> tail;
```

Listing 9.23 (Continued) *GList2Demo.java: a list whose element type is bounded by Comparable, with a recursive sorted method, and a bounded generic method max.*

```
    public GList2() {
        empty = true;
        head = null;
        tail = null;
    }

    public GList2(T head, GList2<T> tail) {
        this.empty = false;
        this.head = head;
        this.tail = tail;
    }

    public boolean empty() {
        return empty;
    }

    public T head() {
        if (empty) {
            throw new IllegalStateException("Trying to access head of an empty list");
        }
        return head;
    }

    public GList2<T> tail() {
        if (empty) {
            throw new IllegalStateException("Trying to access tail of an empty list");
        }
        return tail;
    }

    // Is the list in non-decreasing order? Compare the first two, then recurse.
    public boolean sorted(GList2<T> list) {
        if (list.empty() || list.tail().empty()) {
            return true;
        } else {
            T a = list.head();
            T b = list.tail().head();
            int compareResult = a.compareTo(b);   // allowed only because of the bound
            // 0 means they are equal
            // negative (<0) means a comes before b
            // positive (>0) means a comes after b
            if (compareResult > 0) {
                return false;
            } else {
                return sorted(list.tail());
            }
```

Listing 9.23 (Continued) *GList2Demo.java: a list whose element type is bounded by Comparable, with a recursive sorted method, and a bounded generic method max.*

```
        }
    }

    public boolean sorted() {                    // helper: no argument needed
        return sorted(this);
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("[");
        for (GList2<T> p = this; !p.empty; p = p.tail) {
            sb.append(p.head);
            if (!p.tail.empty) {
                sb.append(", ");
            }
        }
        return sb.append("]").toString();
    }
}
```

```
myGList4: []
myGList4 is sorted: true
myGList5: [5]
myGList5 is sorted: true
myGList6: [10, 5]
myGList6 is sorted: false
myGList7: [3, 5]
myGList7 is sorted: true
words: [iron, wash] is sorted: true
max 9, max wash
```

The method `sorted(list)` is recursive with the two parts of Section 9.2.2. The base case: a list of zero or one elements is sorted. The recursive case: compare the first two elements; if the first comes after the second (`compareResult > 0`), the list is not sorted; otherwise the list is sorted exactly when its tail is. The comments restate the contract of `compareTo` from Chapter 6: zero means equal, negative means `a` comes before `b`, positive means after. The outputs agree: `[10, 5]` is not sorted because $10 > 5$, while `[3, 5]` is. Because the method takes the list as a parameter, the call reads `myGList6.sorted(myGList6)`, which is clumsy; the listing adds the helper `sorted()` that passes `this`, so a user writes `words.sorted()`. (You may meet this method declared as `public <U extends Comparable<U>> boolean sorted(GList2<T> list)`. The `<U …>` declares a second type parameter that the method never uses: harmless, but misleading; the bound on `T` is what makes `compareTo` legal.)

The static method `max` shows a bound on a generic method: `static <T extends Comparable<T>> T max(List<T> items)` works for a list of integers and for a list of strings, and refuses a list of `Object`s at compile time.

To see the bound doing its job, add `GList2<Object> things = new GList2<>();` as the first line of `main`:

```
GList2Demo.java:16: error: type argument Object is not within bounds of type-variable T
        GList2<Object> things = new GList2<>();
               ^
  where T is a type-variable:
    T extends Comparable<T> declared in class GList2
GList2Demo.java:16: error: incompatible types: cannot infer type arguments for GList2<>
        GList2<Object> things = new GList2<>();
                                    ^
    reason: inference variable T has incompatible bounds
      equality constraints: Object
      upper bounds: Comparable<T>
  where T is a type-variable:
    T extends Comparable<T> declared in class GList2
2 errors
```

Python expresses a bound with `TypeVar("T", bound=...)`. The bound here is a **protocol** (Chapter 6): `SupportsLessThan`, any type with a `__lt__` method, which is what the `<` operator calls. **Listing 9.24** shows it.

Listing 9.24 `glist2_demo.py`: a `TypeVar` bounded by the protocol `SupportsLessThan`; mypy rejects a list of plain `object`s, and Python fails when it tries to compare them.

```
"""glist2_demo.py: a bounded type variable - T must support the < operator."""
from __future__ import annotations

from typing import Any, Generic, Protocol, Sequence, TypeVar


class SupportsLessThan(Protocol):
    def __lt__(self, other: Any, /) -> bool: ...


T = TypeVar("T", bound=SupportsLessThan)   # T is bounded


def max_of(items: Sequence[T]) -> T:
    best = items[0]
    for x in items:
        if best < x:
            best = x
    return best


class GList2(Generic[T]):
    _head: T
    _tail: GList2[T]

    def __init__(self) -> None:
        self._empty = True

    @staticmethod
    def nil() -> GList2[T]:
        return GList2()

    @staticmethod
    def cons(head: T, tail: GList2[T]) -> GList2[T]:
        lst: GList2[T] = GList2()
        lst._empty, lst._head, lst._tail = False, head, tail
        return lst

    def empty(self) -> bool:
        return self._empty

    def head(self) -> T:
        if self._empty:
            raise ValueError("Trying to access head of an empty list")
        return self._head
```

Listing 9.24 (Continued) *glist2_demo.py: a TypeVar bounded by the protocol SupportsLessThan; mypy rejects a list of plain objects, and Python fails when it tries to compare them.*

```
    def tail(self) -> GList2[T]:
        if self._empty:
            raise ValueError("Trying to access tail of an empty list")
        return self._tail

    def sorted(self) -> bool:
        """Is the list in non-decreasing order? Compare the first two, then recurse."""
        if self.empty() or self.tail().empty():
            return True
        a, b = self.head(), self.tail().head()
        if b < a:                               # allowed only because of the bound
            return False
        return self.tail().sorted()

    def __str__(self) -> str:
        parts = []
        p = self
        while not p.empty():
            parts.append(str(p.head()))
            p = p.tail()
        return "[" + ", ".join(parts) + "]"


def main() -> None:
    my_glist4: GList2[int] = GList2.nil()
    print(f"myGList4: {my_glist4}")
    print(f"myGList4 is sorted: {my_glist4.sorted()}")
    my_glist5 = GList2.cons(5, my_glist4)
    print(f"myGList5: {my_glist5}")
    print(f"myGList5 is sorted: {my_glist5.sorted()}")
    my_glist6 = GList2.cons(10, my_glist5)
    print(f"myGList6: {my_glist6}")
    print(f"myGList6 is sorted: {my_glist6.sorted()}")
    my_glist7 = GList2.cons(3, my_glist5)
    print(f"myGList7: {my_glist7}")
    print(f"myGList7 is sorted: {my_glist7.sorted()}")

    words = GList2.cons("iron", GList2.cons("wash", GList2.nil()))
    print(f"words: {words} is sorted: {words.sorted()}")
    print(f"max {max_of([3, 9, 4])}, max {max_of(['wash', 'dry', 'iron'])}")
```

Listing 9.24 (Continued) *glist2_demo.py: a TypeVar bounded by the protocol SupportsLessThan; mypy rejects a list of plain objects, and Python fails when it tries to compare them.*

```
    things = GList2.cons(object(), GList2.cons(object(), GList2.nil()))   # mypy: error
    try:
        things.sorted()
    except TypeError as e:
        print("TypeError:", e)


if __name__ == "__main__":
    main()
```

```
myGList4: []
myGList4 is sorted: True
myGList5: [5]
myGList5 is sorted: True
myGList6: [10, 5]
myGList6 is sorted: False
myGList7: [3, 5]
myGList7 is sorted: True
words: [iron, wash] is sorted: True
max 9, max wash
TypeError: '<' not supported between instances of 'object' and 'object'
```

The Python `sorted` is a method of the list itself (`self.tail().sorted()`), so it needs no helper. The last lines build a list of two plain `object()`s. Python runs the construction happily and fails only inside `sorted`, when `<` meets two objects that cannot be compared. mypy objects at the line where the list is built, which is where the mistake is:

```
glist2_demo.py:88: error: Value of type variable "T" of "cons" of "GList2" cannot be
"object"  [type-var]
Found 1 error in 1 file (checked 1 source file)
```

9.8.5 Wildcards: why a List is not a List

Here is a puzzle that every Java programmer meets. `Cat` extends `Animal`, so a `Cat` *is an* `Animal`. Is a `List<Cat>` therefore a `List<Animal>`? It seems obvious. It is false:

```
List<Cat> cats = new ArrayList<Cat>();
List<Animal> animals = cats;          // compile error
```

Suppose Java allowed the second line. `animals` and `cats` would then refer to the *same* list object, and since `animals` has type `List<Animal>`, this line would compile too:

```
animals.add(new Dog("wondergirl"));   // a Dog is an Animal, so why not?
```

Now the list that `cats` promises contains only cats holds a dog, and the next `Cat c = cats.get(2)` fails. The only safe rule is that `List<Cat>` and `List<Animal>` are unrelated types, even though `Cat` and `Animal` are related. In general, for any generic type G:

$$Cat <: Animal \quad \not\Rightarrow \quad G\langle Cat\rangle <: G\langle Animal\rangle \tag{9.7}$$

where $<:$ is read “is a subtype of”. Generic types that obey Equation (9.7) are called **invariant**. **Figure 9.8** draws the situation, including the two escape routes that Java provides, the **wildcards**.

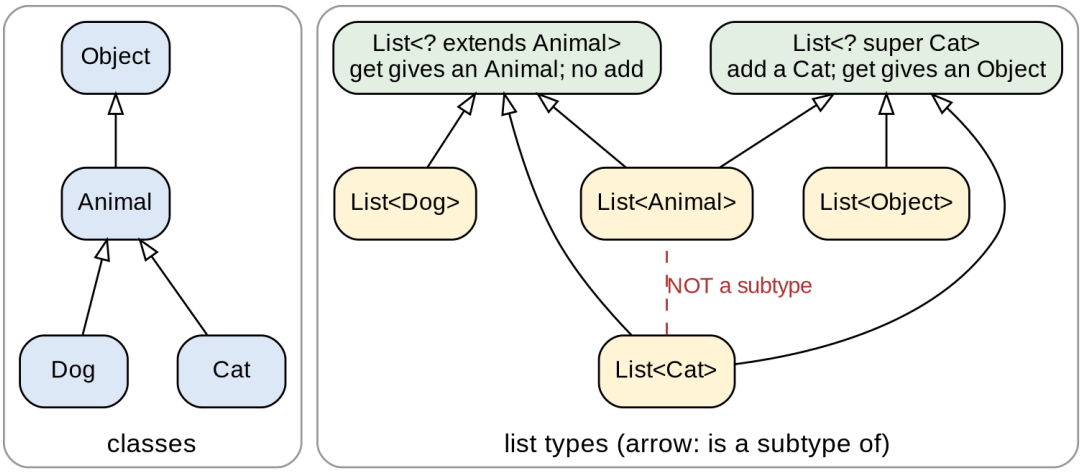


Figure 9.8 Subtyping among classes (left) and among list types (right). `List<Cat>` is not a subtype of `List<Animal>`, but both are subtypes of `List<? extends Animal>`, and `List<Cat>`, `List<Animal>` and `List<Object>` are all subtypes of `List<? super Cat>`.

A wildcard `?` stands for "some type I do not name". It comes in two bounded forms:

- `List<? extends Animal>` means "a list of *some* subtype of `Animal`" — maybe `List<Animal>`, maybe `List<Cat>`, maybe `List<Dog>`. Whatever it is, every element is an `Animal`, so **reading** is safe: `get` returns an `Animal`. But **adding** is forbidden, because the compiler does not know which subtype the list really holds; it might be a list of cats, and the new element might be a dog. The only value you can add is `null`.
- `List<? super Cat>` means "a list of *some* supertype of `Cat`" — `List<Cat>`, `List<Animal>` or `List<Object>`. Whatever it is, it can hold a `Cat`, so **adding** a `Cat` is safe. But reading gives only an `Object`, because the compiler does not know which supertype it is.

Joshua Bloch's mnemonic is **PECS: producer `extends`, consumer `super`** [2]. A parameter from which the method only *takes* values produces them, and should be `? extends T`; a parameter into which the method only *puts* values consumes them, and should be `? super T`. **Table 9.3** summarizes.

Table 9.3 What each kind of list declaration may hold, and what it lets you read and write.

Declaration	May hold	`get` returns	`add` accepts	Use it for
`List<Animal>`	exactly a `List<Animal>`	`Animal`	`Animal` (and subclasses)	reading and writing
`List<? extends Animal>`	`List<Animal>`, `List<Cat>`, `List<Dog>`	`Animal`	nothing (only `null`)	a *producer*: read only
`List<? super Cat>`	`List<Cat>`, `List<Animal>`, `List<Object>`	`Object`	`Cat` (and subclasses)	a *consumer*: write only
`List<?>`	a list of anything	`Object`	nothing (only `null`)	size, clear, printing

Listing 9.25 makes all of this concrete with the classes `Animal`, `Cat` and `Dog`.

Listing 9.25 `WildcardDemo.java`: `Animal`, `Cat` and `Dog`; a reader method with `? extends Animal`; the method `addAll(src, dest)` with `? super T`; and the covariant array that fails at run time.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class WildcardDemo {
    // Reads from src (a producer of T) and writes into dest (a consumer of T).
    public static <T> void addAll(List<T> src, List<? super T> dest) {
        for (T element : src) {        // add every element in src to dest
            dest.add(element);
        }
    }

    // Accepts a List<Animal>, a List<Cat> or a List<Dog>: it only reads.
    static String names(List<? extends Animal> animals) {
        StringBuilder sb = new StringBuilder();
        for (Animal a : animals) {
            sb.append(a.getName()).append(' ');
        }
        return sb.toString().trim();
    }

    public static void main(String[] args) {
        Cat cat1 = new Cat();
        cat1.setName("superman");
        Cat cat2 = new Cat();
        cat2.setName("spiderman");
        List<Cat> cats = new ArrayList<Cat>();
        cats.add(cat1);
        cats.add(cat2);
        Iterator<Cat> iterator = cats.iterator();
        while (iterator.hasNext()) {
            Cat c = iterator.next();                    // no cast: the iterator is typed
            System.out.println(c.getName());
        }

        Dog dog1 = new Dog();
        dog1.setName("supergirl");
        Dog dog2 = new Dog();
        dog2.setName("spidergirl");
        List<Dog> dogs = new ArrayList<>();
        dogs.add(dog1);
        dogs.add(dog2);
        System.out.println("dogs: " + names(dogs));
```

Listing 9.25 (Continued) *WildcardDemo.java: Animal, Cat and Dog; a reader method with ? extends Animal; the method addAll(src, dest) with ? super T; and the covariant array that fails at run time.*

```
        // List<Animal> zoo = cats;                    // compile error: see the text
        List<? extends Animal> animals = cats;         // read-only view as Animals
        Animal animal1 = animals.get(0);
        System.out.println(animal1.getName() + " / " + animals.get(1).getName()
                + " / " + animal1);
        // animals.add(dog1);                          // compile error: see the text

        List<Animal> newAnimals = new ArrayList<>();
        addAll(cats, newAnimals);                      // T = Cat, dest is List<? super Cat>
        addAll(dogs, newAnimals);                      // T = Dog
        System.out.println("newAnimals: " + names(newAnimals));
        List<Object> things = new ArrayList<>();
        addAll(dogs, things);                          // Object is a supertype of Dog too
        System.out.println("things: " + things);

        // Arrays are covariant, so the same mistake is caught only at run time.
        Animal[] pets = new Cat[2];
        try {
            pets[0] = dog1;
        } catch (ArrayStoreException e) {
            System.out.println("ArrayStoreException: " + e.getMessage());
        }
    }
}

class Animal {
    private String name;

    public Animal() {
    }

    public Animal(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

Listing 9.25 (Continued) *WildcardDemo.java: Animal, Cat and Dog; a reader method with ? extends Animal; the method addAll(src, dest) with ? super T; and the covariant array that fails at run time.*

```
    @Override
    public String toString() {
        return "This is Animal class";
    }
}

class Cat extends Animal {
    public Cat() {
    }

    @Override
    public String toString() {
        return "This is Cat class";
    }
}

class Dog extends Animal {
    public Dog() {
    }

    public Dog(String name) {
        super(name);
    }

    @Override
    public String toString() {
        return "This is Dog class";
    }
}
```

```
superman
spiderman
dogs: supergirl spidergirl
superman / spiderman / This is Cat class
newAnimals: superman spiderman supergirl spidergirl
things: [This is Dog class, This is Dog class]
ArrayStoreException: Dog
```

The typed `Iterator<Cat>` needs no cast. The method `names(List<? extends Animal>)` accepts the list of dogs here and the list of cats and the mixed list later: it only reads, so `? extends` is exactly right. The assignment `List<? extends Animal> animals = cats` is legal, and `animals.get(0)` gives an `Animal` whose name is `superman`; printing it runs `Cat`'s `toString`, *This is Cat class*, by dynamic dispatch (Chapter 7).

Remove the `//` in front of the two lines marked "compile error" and `javac` explains both rules in its own words:

```
WildcardDemo.java:45: error: incompatible types: List<Cat> cannot be converted to
List<Animal>
        List<Animal> zoo = cats;
                           ^
WildcardDemo.java:50: error: incompatible types: Dog cannot be converted to CAP#1
        animals.add(dog1);
                    ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Animal from capture of ? extends Animal
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
2 errors
```

In the second message, `CAP#1` is the compiler's name for the unknown type behind this `?`; all it knows is that `CAP#1` extends `Animal`, and a `Dog` might not be one.

The method `addAll(List<T> src, List<? super T> dest)` copies every element of `src` into `dest`. `src` is a producer and could even be `List<? extends T>`; `dest` is a consumer, `? super T`. With `T = Cat`, `dest` may be a `List<Animal>`; with `T = Dog`, the same `List<Animal>` again, which now holds two cats and two dogs; and a `List<Object>` would do as well. The Java library's own `Collections.addAll` and `Collections.copy` are declared in exactly this way.

The last block explains why Java's designers were so strict with generics: they had already been bitten by **arrays**, which are **covariant** — a `Cat[]` *is* an `Animal[]`. So `Animal[] pets = new Cat[2]` compiles, and `pets[0] = dog1` compiles too. The array remembers at run time that it is really a `Cat[]`, and the store fails with `ArrayStoreException: Dog`. Generics catch the same mistake at compile time instead.

Python's type checker applies the same reasoning, and **Listing 9.26** lets us watch what happens when nobody checks.

Listing 9.26 `wildcard_demo.py`: `Sequence[Animal]` accepts a list of cats (read-only, covariant); `list[Animal]` does not (invariant), and when the rule is ignored a dog ends up among the cats.

```
"""wildcard_demo.py: variance in Python - list is invariant, Sequence is covariant."""
from __future__ import annotations

from typing import Iterable, Sequence, TypeVar

T = TypeVar("T")


class Animal:
    def __init__(self, name: str = "") -> None:
        self.name = name

    def __str__(self) -> str:
        return "This is Animal class"


class Cat(Animal):
    def __str__(self) -> str:
        return "This is Cat class"


class Dog(Animal):
    def __str__(self) -> str:
        return "This is Dog class"


def add_all(src: Iterable[T], dest: list[T]) -> None:
    """Reads from src and appends to dest."""
    for element in src:
        dest.append(element)


def names(animals: Sequence[Animal]) -> str:
    """Accepts a list of Animal, of Cat or of Dog: Sequence is read-only."""
    return " ".join(a.name for a in animals)


def main() -> None:
    cats: list[Cat] = [Cat("superman"), Cat("spiderman")]
    for c in cats:
        print(c.name)
    dogs: list[Dog] = [Dog("supergirl"), Dog("spidergirl")]
    print("dogs:", names(dogs))
```

Listing 9.26 (Continued) *wildcard_demo.py: Sequence[Animal] accepts a list of cats (read-only, covariant); list[Animal] does not (invariant), and when the rule is ignored a dog ends up among the cats.*

```
    animals: Sequence[Animal] = cats              # fine: a read-only view
    print(f"{animals[0].name} / {animals[1].name} / {animals[0]}")

    new_animals: list[Animal] = []
    add_all(cats, new_animals)
    add_all(dogs, new_animals)
    print("newAnimals:", names(new_animals))
    things: list[object] = []
    add_all(dogs, things)
    print("things:", [str(t) for t in things])

    # What Java forbids at compile time, Python allows at run time.
    zoo: list[Animal] = cats                      # mypy: error (list is invariant)
    zoo.append(Dog("wondergirl"))
    print("cats now holds:", [str(c) for c in cats])


if __name__ == "__main__":
    main()
```

```
superman
spiderman
dogs: supergirl spidergirl
superman / spiderman / This is Cat class
newAnimals: superman spiderman supergirl spidergirl
things: ['This is Dog class', 'This is Dog class']
cats now holds: ['This is Cat class', 'This is Cat class', 'This is Dog class']
```

In Python's typing, `list[T]` is invariant, exactly like Java's `List<T>`, and `Sequence[T]` — a *read-only* view with no `append` — is **covariant**: a `list[Cat]` may be passed where a `Sequence[Animal]` is expected. So `Sequence[Animal]` plays the part of `List<? extends Animal>`. For the consumer side Python needs no `? super`: the function `add_all(src: Iterable[T], dest: list[T])` solves `T = Animal` for the call with a `list[Animal]` destination, and accepts `cats` as an `Iterable[Animal]` because `Iterable` is covariant too. The last three lines of `main` do what Java refused: `zoo: list[Animal] = cats`, then `zoo.append(Dog("wondergirl"))`. Python runs it, and the output shows the result: the list of cats now holds a dog. mypy points at the guilty line, and even suggests the fix:

```
wildcard_demo.py:57: error: Incompatible types in assignment (expression has type
"list[Cat]", variable has type "list[Animal]")  [assignment]
wildcard_demo.py:57: note: "list" is invariant -- see
https://mypy.readthedocs.io/en/stable/common_issues.html#variance
wildcard_demo.py:57: note: Consider using "Sequence" instead, which is covariant
Found 1 error in 1 file (checked 1 source file)
```

9.8.6 Type erasure: what is left at run time

How does the Java virtual machine run `List<String>` and `List<Integer>`? It doesn't. Generics were added to Java in 2004, twelve years after the virtual machine was designed, and had to work with existing class files and libraries. So Java implements them by **type erasure**: the compiler checks every use of type arguments, inserts the casts that the old raw code had to write by hand, and then *erases* the type arguments. In the class file, `List<String>` is just `List`, and a type parameter `T` becomes its bound, `Object` or `Comparable` [1], [13]. **Listing 9.27** shows the consequences.

Listing 9.27 `ErasureDemo.java`: at run time a `List<String>` and a `List<Integer>` are the same class; a raw reference can smuggle an `Integer` into a `List<String>`, and `Collections.checkedList` stops it at the door.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ErasureDemo {
    @SuppressWarnings({"rawtypes", "unchecked"})
    static void sneakIn(List list) {       // a raw parameter: the compiler stops checking
        list.add(42);
    }

    public static void main(String[] args) {
        List<String> strings = new ArrayList<>();
        List<Integer> numbers = new ArrayList<>();
        System.out.println("same class at run time: "
                + (strings.getClass() == numbers.getClass()));
        System.out.println("class name: " + strings.getClass().getName());

        // Heap pollution: an Integer inside a List<String>.
        strings.add("Siti");
        sneakIn(strings);                  // nothing fails here ...
        System.out.println("size " + strings.size() + ", contents " + strings);
        try {
            String s = strings.get(1);     // ... the failure appears here, far away
            System.out.println(s);
        } catch (ClassCastException e) {
            System.out.println("ClassCastException at get(1)");
        }

        // A checked view fails at the moment of the bad add instead.
        List<String> guarded = Collections.checkedList(new ArrayList<>(), String.class);
        guarded.add("Budi");
        try {
            sneakIn(guarded);
        } catch (ClassCastException e) {
            System.out.println("checkedList refused: " + e.getMessage());
        }
    }
}
```

```
same class at run time: true
class name: java.util.ArrayList
size 2, contents [Siti, 42]
ClassCastException at get(1)
checkedList refused: Attempt to insert class java.lang.Integer element into collection with
element type class java.lang.String
```

The first two lines prove the point: both lists are simply `java.util.ArrayList` at run time. The next block shows **heap pollution**. The method `sneakIn` takes a *raw* `List`, which switches off checking inside it, and adds `42` to what the caller believes is a `List<String>`. Nothing fails at the `add`, because at run time there is no `String` in the list's type to check against. The list now prints as `[Siti, 42]`. The failure appears later, at `strings.get(1)`, where the compiler inserted a cast to `String` on our behalf. This is why raw types and "unchecked" warnings deserve suspicion: they are the only way to break the guarantee of **Listing 9.17**. When you must pass a collection to code you do not trust, wrap it with `Collections.checkedList(list, String.class)`: the checked view *remembers* the element type and rejects a wrong element at the moment it is added, as the last line shows.

Erasure also explains four things you cannot write in Java. Given the file `Limits.java`

```
import java.util.List;

class Limits<T> {
    T make() { return new T(); }
    T[] many() { return new T[10]; }
    boolean test(Object o) { return o instanceof List<String>; }
    List<int> numbers;
}
```

`javac` answers:

```
Limits.java:7: error: unexpected type
    List<int> numbers;
         ^
  required: reference
  found:    int
Limits.java:4: error: unexpected type
    T make() { return new T(); }
                         ^
  required: class
  found:    type parameter T
  where T is a type-variable:
    T extends Object declared in class Limits
Limits.java:5: error: generic array creation
    T[] many() { return new T[10]; }
                        ^
Limits.java:6: error: Object cannot be safely cast to List<String>
    boolean test(Object o) { return o instanceof List<String>; }
                                    ^
4 errors
```

At run time there is no `T` to construct or to make an array of (the remedy is to pass in a factory or a `Class<T>`; `ArrayList` itself keeps an `Object[]`), no `String` left to check with `instanceof` (`instanceof List<?>` is allowed), and type arguments must be reference types, so numbers go in a `List<Integer>`, with boxing (Chapter 5).

Python goes further than erasure: type arguments were *never* part of the running program. **Listing 9.28** shows what remains.

Listing 9.28 `erasure_demo.py`: `list[str]` is only an alias for `list`; nothing checks type arguments at run time, but the hints stay available to tools.

```
"""erasure_demo.py: type arguments in Python are for tools, not for the running program."""
from typing import Generic, TypeVar, get_origin, get_type_hints

T = TypeVar("T")


class Box(Generic[T]):
    def __init__(self, item: T) -> None:
        self.item = item


class Shelf:
    boxes: list[Box[int]]


def main() -> None:
    strings: list[str] = []
    numbers: list[int] = []
    print("same class at run time:", type(strings) is type(numbers))
    print("class name:", type(strings).__name__)

    alias = list[str]
    print("list[str] is", repr(alias), "- its origin is", get_origin(alias))
    made = list[str]([1, 2])           # calling the alias just makes a plain list
    print("list[str]([1, 2]) =", made)
    try:
        isinstance(strings, list[str])
    except TypeError as e:
        print("TypeError:", e)

    b = Box[int]("not an int")         # nothing checks the argument
    print("Box[int] holds", repr(b.item), "- remembered:", getattr(b, "__orig_class__"))
    print("hints kept for tools:", get_type_hints(Shelf))


if __name__ == "__main__":
    main()
```

```
same class at run time: True
class name: list
list[str] is list[str] - its origin is <class 'list'>
list[str]([1, 2]) = [1, 2]
TypeError: isinstance() argument 2 cannot be a parameterized generic
Box[int] holds 'not an int' - remembered: __main__.Box[int]
hints kept for tools: {'boxes': list[__main__.Box[int]]}
```

`list[str]` is a small alias object whose *origin* is `list`; calling it makes an ordinary list, even of integers. `isinstance(x, list[str])` is refused with `TypeError`, for the same reason as Java's `instanceof List<String>`. A `Box[int]` happily holds a string, though the object does remember, in the attribute `__orig_class__`, which alias created it. And the hints written in the program are kept as data, for tools: `typing.get_type_hints(Shelf)` returns them. mypy, of course, reports every one of the deliberate mistakes in this file (four errors, on lines 24, 27 and 31).

9.8.7 Generics in Java and Python: a summary

Table 9.4 collects the vocabulary of this section in both languages.

Table 9.4 Generic programming in Java and in Python.

Idea	Java	Python (checked by mypy)
type parameter of a class	`class Box<T>`	`T = TypeVar("T")`, `class Box(Generic[T])`; since 3.12 `class Box[T]:`
type argument	`Box<Integer>`	`Box[int]`
generic method or function	`static <T> T first(List<T> xs)`	`def first(xs: Sequence[T]) -> T`
bounded parameter	`<T extends Comparable<T>>`	`TypeVar("T", bound=SupportsLessThan)`
built-in collection types	`List<String>`, `Map<String, Integer>`	`list[str]`, `dict[str, int]` (3.9+)
read-only, covariant view	`List<? extends Animal>`	`Sequence[Animal]`, `Iterable[Animal]`
write-only, contravariant	`List<? super Cat>`	rarely needed; a `TypeVar` solved at the call
unknown element type	`List<?>`	`list[Any]`, `Sequence[object]`
what the running program sees	erased: `List`	nothing: `list`
who enforces it	`javac`, always	mypy or another checker, when you run it

9.9 Putting Things in Order: Comparable and Comparator

A list can be sorted with `Collections.sort(list)` or `list.sort(...)`, but not every list can be sorted: the sort must be able to ask of any two elements, "which comes first?". There are exactly two ways to answer:

1. The element class implements **`Comparable<T>`**, whose single method `compareTo(T other)` defines the class's *natural ordering* (Chapter 6). `String`, `Integer` and `LocalDate` all do. Then `Collections.sort(list)` needs nothing else.
2. We write a **`Comparator<T>`**, a separate object whose method `compare(a, b)` defines *an* ordering, chosen from outside the class, and pass it to the sort: `Collections.sort(list, comparator)`. A class has one natural ordering but may have any number of comparators: by age, by name, by visits, largest first.

Both methods follow the same contract: the result is negative if the first argument comes first, zero if the two are equal in this ordering, positive if the first comes after [5]. A correct ordering must also be *consistent*: if a comes before b and b before c, then a before c; otherwise sorting may fail with `IllegalArgumentException: Comparison method violates its general contract!`.

9.9.1 Iterators, duplicates, iterables and a comparator

Listing 9.29 is a tour in five parts that ties together several ideas of this chapter, ending with a `Comparator` for students.

Listing 9.29 `MyCollection.java`: an iterator, the fact that an iterator only reads, removing duplicates from a sorted list, an `Iterable` in a for-each loop, and sorting students with `AgeComparator`.

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;

public class MyCollection {
    public static <T> List<T> removeDuplicatesFromSortedList(List<T> input) {
        List<T> result = new ArrayList<>();
        if (input.isEmpty()) {
            return result;
        }
        T previous = null;
        result.add(input.get(0));
        Iterator<T> iter = input.iterator();
        while (iter.hasNext()) {
            T current = iter.next();
            if (previous != null && !previous.equals(current)) {
                result.add(current);
            }
            previous = current;
        }
        return result;
    }

    public static void main(String[] args) {
        System.out.println("Part 1: Iterator");
        Collection<String> myCollection = new ArrayList<>();
        myCollection.add("cat");
        Iterator<String> iterator = myCollection.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        System.out.printf("%nPart 2: Iterator is only read not change%n");
        Iterator<String> iterator2 = myCollection.iterator();
        while (iterator2.hasNext()) {
            String element = iterator2.next();   // there is no iterator2.prev()
            element += " and dog";                // changes the local copy only
            System.out.println(element);
        }
        System.out.println("collection still holds " + myCollection);
```

Listing 9.29 (Continued) *MyCollection.java: an iterator, the fact that an iterator only reads, removing duplicates from a sorted list, an Iterable in a for-each loop, and sorting students with AgeComparator.*

```
        System.out.printf("%nPart 3: Removing duplicate values (sorted list only)%n");
        myCollection.add("cat");
        myCollection.add("dog");
        myCollection.add("monkey");
        myCollection.add("monkey");
        Iterator<String> iterator3 = myCollection.iterator();
        while (iterator3.hasNext()) {
            System.out.println(iterator3.next());
        }
        System.out.printf("%nAfter removing the duplicate values%n");
        Collection<String> myCollection2 =
                removeDuplicatesFromSortedList((List<String>) myCollection);
        Iterator<String> iterator4 = myCollection2.iterator();
        while (iterator4.hasNext()) {
            System.out.println(iterator4.next());
        }

        System.out.printf("%nPart 4: Iterable%n");
        Collection<Integer> it = new ArrayList<>();
        it.add(3);
        it.add(5);
        it.add(7);
        int sum = 0;
        for (Integer i : it) {                    // works for anything Iterable
            sum += i;
        }
        System.out.println("sum = " + sum);

        System.out.printf("%nPart 5: Comparator%n");
        Collection<Student> students = new ArrayList<>();
        Student s = new Student(15);
        students.add(s);
        Student s1 = new Student(20);
        students.add(s1);
        Student s2 = new Student(12);
        students.add(s2);
        Iterator<Student> itStudents = students.iterator();
        while (itStudents.hasNext()) {
            System.out.println(itStudents.next().getAge());
        }
        AgeComparator age = new AgeComparator();
        System.out.printf("Age difference between s1 (age %d) and s2 (age %d) = %d%n",
                s1.getAge(), s2.getAge(), age.compare(s1, s2));
        System.out.println("After sort is called, the list of student becomes:");
        Collections.sort((List<Student>) students, new AgeComparator());
        Iterator<Student> itStudents2 = students.iterator();
```

Listing 9.29 (Continued) *MyCollection.java: an iterator, the fact that an iterator only reads, removing duplicates from a sorted list, an Iterable in a for-each loop, and sorting students with AgeComparator.*

```
        while (itStudents2.hasNext()) {
            System.out.println(itStudents2.next().getAge());
        }
    }
}

class Student {
    private final int age;

    Student(int age) {
        this.age = age;
    }

    public int getAge() {
        return age;
    }
}

// Using list -> sort all the elements in order, by Collections.sort().
// To sort a list of objects: (1) the class implements Comparable, or
// (2) we write a Comparator class for the objects we want to sort.
class AgeComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        if (s1 == s2) {
            return 0;               // the same object (or both null)
        } else if (s1 == null) {
            return -1;              // null comes first
        } else if (s2 == null) {
            return 1;
        } else {
            return s1.getAge() - s2.getAge();   // younger first
        }
    }
}
```

```
Part 1: Iterator
cat

Part 2: Iterator is only read not change
cat and dog
collection still holds [cat]

Part 3: Removing duplicate values (sorted list only)
cat
cat
dog
monkey
monkey

After removing the duplicate values
cat
dog
monkey

Part 4: Iterable
sum = 15

Part 5: Comparator
15
20
12
Age difference between s1 (age 20) and s2 (age 12) = 8
After sort is called, the list of student becomes:
12
15
20
```

Part 1 walks a one-element collection with an explicit `Iterator`. Note the declared type `Collection<String>`, the root interface of **Figure 9.2**: for this part we need nothing more than a group of elements.

Part 2 surprises many beginners: the loop prints `cat and dog`, but the collection still holds `[cat]`. `element` received a *reference* to `"cat"`; `+=` builds a new string and points the local variable at it, leaving the collection's reference untouched. To change an element of a list, call `set` (or `ListIterator.set`; a `ListIterator` can also move backwards, which a plain `Iterator` cannot).

Part 3 removes duplicates from `[cat, cat, dog, monkey, monkey]` with the original form of the method of Section 9.8.3, now guarded by an `isEmpty` test. The cast `(List<String>) myCollection` is needed because the variable was declared as a `Collection`; declaring the interface the code really needs, `List`, would avoid it.

Part 4 shows that the for-each loop works on anything `Iterable`: $3 + 5 + 7 = 15$.

Part 5 sorts three students by age with `AgeComparator`. Look at its `compare`. `s1 == s2` catches both "the same object" and "both `null`". The next two tests make `null` sort first. The last line, `s1.getAge() - s2.getAge()`, is the classic shortcut: the difference is negative when `s1` is younger, zero when equal, positive when older. Printed for `s1` (age 20) and `s2` (age 12), it is $20 - 12 = 8$, positive, so `s1` comes after `s2`. After `Collections.sort` the ages come out `12 15 20`. The subtraction trick is safe for ages, but Section 9.9.2 shows it failing for large numbers.

Listing 9.30 tells the same story in Python.

Listing 9.30 `my_collection.py`: the five parts in Python; a Java-style comparison function adapted with `functools.cmp_to_key`, and the Pythonic `key=` function.

```
"""my_collection.py: iterators, duplicates, iterables and comparators in Python."""
from functools import cmp_to_key


def remove_duplicates_from_sorted_list(items):
    result = []
    previous = None
    for current in items:
        if not result or previous != current:
            result.append(current)
        previous = current
    return result


class Student:
    def __init__(self, age):
        self._age = age

    def get_age(self):
        return self._age


def age_compare(s1, s2):
    """A comparison function in the style of Java's Comparator.compare."""
    if s1 is s2:
        return 0
    if s1 is None:
        return -1                       # None comes first
    if s2 is None:
        return 1
    return s1.get_age() - s2.get_age()     # younger first


def main():
    print("Part 1: Iterator")
    my_collection = ["cat"]
    iterator = iter(my_collection)
    for element in iterator:
        print(element)

    print("\nPart 2: Iterator is only read not change")
    for element in my_collection:
        element += " and dog"           # rebinds the loop variable only
        print(element)
    print("collection still holds", my_collection)
```

Listing 9.30 (Continued) *my_collection.py: the five parts in Python; a Java-style comparison function adapted with functools.cmp_to_key, and the Pythonic key= function.*

```
    print("\nPart 3: Removing duplicate values (sorted list only)")
    my_collection += ["cat", "dog", "monkey", "monkey"]
    for element in my_collection:
        print(element)
    print("\nAfter removing the duplicate values")
    for element in remove_duplicates_from_sorted_list(my_collection):
        print(element)

    print("\nPart 4: Iterable")
    it = [3, 5, 7]
    total = 0
    for i in it:                        # works for anything iterable
        total += i
    print("sum =", total)

    print("\nPart 5: Comparator")
    s, s1, s2 = Student(15), Student(20), Student(12)
    students = [s, s1, s2]
    for st in students:
        print(st.get_age())
    print(f"Age difference between s1 (age {s1.get_age()}) and s2 (age {s2.get_age()}) "
          f"= {age_compare(s1, s2)}")
    print("After sort is called, the list of student becomes:")
    students.sort(key=cmp_to_key(age_compare))     # old-style comparator
    for st in students:
        print(st.get_age())
    print("with key=:", [st.get_age() for st in sorted(students, key=Student.get_age)])


if __name__ == "__main__":
    main()
```

```
Part 1: Iterator
cat

Part 2: Iterator is only read not change
cat and dog
collection still holds ['cat']

Part 3: Removing duplicate values (sorted list only)
cat
cat
dog
monkey
monkey

After removing the duplicate values
cat
dog
monkey

Part 4: Iterable
sum = 15

Part 5: Comparator
15
20
12
Age difference between s1 (age 20) and s2 (age 12) = 8
After sort is called, the list of student becomes:
12
15
20
with key=: [12, 15, 20]
```

Part 2 behaves exactly as in Java, for the same reason: `element += " and dog"` rebinds the loop variable, and the list is unchanged. Part 5 contains the most important difference in the whole section. Python's sort does not take a two-argument comparison function; it takes a **key function** of *one* argument, and sorts the elements by the values the key returns. `sorted(students, key=Student.get_age)` means "sort by age". A Java-style comparator can still be used by wrapping it, `key=cmp_to_key(age_compare)`, which is handy when porting code, but a key is simpler to write, harder to get wrong, and faster, because each element's key is computed once rather than once per comparison.

9.9.2 Comparators for real use

Since Java 8, comparators are rarely written as classes. The interface `Comparator` has static and default methods that *build* comparators from the field you want to sort by, and combine them. **Listing 9.31** sorts five customers of the laundry in several ways.

Listing 9.31 `SortingDemo.java`: natural order, `comparingInt(...).reversed()`, `comparing(...).thenComparing(...)`, a lambda comparator, a `TreeSet` with a comparator, and the overflow in a subtraction comparator.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.TreeSet;

public class SortingDemo {
    public static void main(String[] args) {
        List<Customer> customers = new ArrayList<>(List.of(
                new Customer("Rina", "Sidoarjo", 4),
                new Customer("Agus", "Surabaya", 7),
                new Customer("Sari", "Surabaya", 4),
                new Customer("Budi", "Gresik", 9),
                new Customer("Dewi", "Gresik", 4)));

        customers.sort(null);                          // natural order: compareTo
        System.out.println("by name      " + customers);

        customers.sort(Comparator.comparingInt(Customer::getVisits).reversed());
        System.out.println("most visits  " + customers);   // ties keep name order

        customers.sort(Comparator.comparing(Customer::getCity)
                .thenComparing(Customer::getName, Comparator.reverseOrder()));
        System.out.println("by city, then name Z-A:");
        for (Customer c : customers) {
            System.out.printf("  %-9s %s%n", c.getCity(), c);
        }

        // A Comparator written as a lambda: (a, b) -> a negative, zero or positive int.
        Comparator<Customer> byVisits = (a, b) -> Integer.compare(a.getVisits(),
b.getVisits());
        System.out.println("fewest visits " + Collections.min(customers, byVisits));

        // A TreeSet with a Comparator: "equal" by the comparator means duplicate!
        TreeSet<Customer> oneAtEachCount = new TreeSet<>(
                Comparator.comparingInt(Customer::getVisits));
        oneAtEachCount.addAll(customers);
        System.out.println("TreeSet by visits " + oneAtEachCount);
```

Listing 9.31 (Continued) *SortingDemo.java: natural order, comparingInt(...).reversed(), comparing(...).thenComparing(...), a lambda comparator, a TreeSet with a comparator, and the overflow in a subtraction comparator.*

```
        // Why subtraction is a dangerous comparator.
        Comparator<Integer> bySubtraction = (a, b) -> a - b;
        int low = -2_000_000_000;
        int high = 2_000_000_000;
        System.out.println("subtraction says low vs high = " + bySubtraction.compare(low,
high)
                + ", Integer.compare says " + Integer.compare(low, high));
    }
}

class Customer implements Comparable<Customer> {
    private final String name;
    private final String city;
    private final int visits;

    Customer(String name, String city, int visits) {
        this.name = name;
        this.city = city;
        this.visits = visits;
    }

    String getName() {
        return name;
    }

    String getCity() {
        return city;
    }

    int getVisits() {
        return visits;
    }

    @Override
    public int compareTo(Customer other) {           // the natural order: by name
        return name.compareTo(other.name);
    }

    @Override
    public String toString() {
        return name + "(" + visits + ")";
    }
}
```

```
by name      [Agus(7), Budi(9), Dewi(4), Rina(4), Sari(4)]
most visits  [Budi(9), Agus(7), Dewi(4), Rina(4), Sari(4)]
by city, then name Z-A:
  Gresik    Dewi(4)
  Gresik    Budi(9)
  Sidoarjo  Rina(4)
  Surabaya  Sari(4)
  Surabaya  Agus(7)
fewest visits Dewi(4)
TreeSet by visits [Dewi(4), Agus(7), Budi(9)]
subtraction says low vs high = 294967296, Integer.compare says -1
```

`customers.sort(null)` sorts by the natural order, `compareTo`, which compares names. `Comparator.comparingInt(Customer::getVisits)` builds a comparator that compares the visits, and `.reversed()` flips it, so the most frequent customer comes first. Now look at the three customers with four visits: `Dewi`, `Rina`, `Sari`, still in name order. That is not luck. Java's `List.sort` for objects is **stable**: elements that compare as equal keep their previous relative order. Stability

is what makes "sort by visits, and among equal visits keep the alphabetical order" possible with a single sort after an earlier one [14].

`Comparator.comparing(Customer::getCity).thenComparing(Customer::getName, Comparator.reverseOrder())` sorts by city and, within a city, by name *descending*: Gresik before Sidoarjo before Surabaya, and within Gresik `Dewi` before `Budi`. The lambda `(a, b) -> Integer.compare(a.getVisits(), b.getVisits())` is a comparator written by hand; `Collections.min` returns the first customer with the fewest visits.

The `TreeSet` line contains a trap. A `TreeSet` built with a comparator uses *that comparator* to decide equality, not `equals`. Three customers have four visits, so the comparator calls them equal, and the set keeps only the first of them. Only three customers survive: `[Dewi(4), Agus(7), Budi(9)]`. Use a comparator with a sorted set or map only if "equal under the comparator" really means "the same element" — typically by adding a tie-breaker such as `.thenComparing(Customer::getName)`.

The last line shows why `AgeComparator`'s subtraction is dangerous in general. For $a = -2{,}000{,}000{,}000$ and $b = 2{,}000{,}000{,}000$, the true difference $a - b = -4 \times 10^9$ does not fit in an `int` (whose range is about $\pm 2.1 \times 10^9$). It wraps around to $-4{,}000{,}000{,}000 + 2^{32} = 294{,}967{,}296$, a *positive* number, so the comparator claims that -2 billion is larger than $+2$ billion. `Integer.compare(a, b)` compares without subtracting and always returns the right sign. Use it, or `Comparator.comparingInt`, always.

Listing 9.32 is the Python version.

Listing 9.32 `sorting_demo.py`: natural order with `__lt__` and `total_ordering`, key functions, two stable sorts for mixed directions, and Python's unbounded integers.

```
"""sorting_demo.py: sorting with key functions, and why Python's sort is stable."""
from functools import total_ordering
from operator import attrgetter


@total_ordering
class Customer:
    def __init__(self, name, city, visits):
        self.name = name
        self.city = city
        self.visits = visits

    def __eq__(self, other):
        return isinstance(other, Customer) and self.name == other.name

    def __hash__(self):
        return hash(self.name)

    def __lt__(self, other):           # the natural order: by name
        return self.name < other.name

    def __repr__(self):
        return f"{self.name}({self.visits})"


def main():
    customers = [Customer("Rina", "Sidoarjo", 4), Customer("Agus", "Surabaya", 7),
                 Customer("Sari", "Surabaya", 4), Customer("Budi", "Gresik", 9),
                 Customer("Dewi", "Gresik", 4)]

    customers.sort()                                    # natural order: __lt__
    print("by name     ", customers)

    customers.sort(key=attrgetter("visits"), reverse=True)
    print("most visits ", customers)                    # ties keep name order

    # Two stable sorts: first the minor key, then the major key.
    customers.sort(key=lambda c: c.name, reverse=True)
    customers.sort(key=lambda c: c.city)
    print("by city, then name Z-A:")
    for c in customers:
        print(f"  {c.city:<9} {c}")

    print("fewest visits", min(customers, key=lambda c: c.visits))
```

Listing 9.32 (Continued) *sorting_demo.py: natural order with __lt__ and total_ordering, key functions, two stable sorts for mixed directions, and Python's unbounded integers.*

```
    # "One customer per visit count", like the TreeSet with a Comparator.
    one_at_each_count = {}
    for c in customers:
        one_at_each_count.setdefault(c.visits, c)      # keep the first one seen
    print("one per count", [one_at_each_count[k] for k in sorted(one_at_each_count)])

    low, high = -2_000_000_000, 2_000_000_000
    print("subtraction says low vs high =", low - high, "(Python ints never overflow)")


if __name__ == "__main__":
    main()
```

```
by name      [Agus(7), Budi(9), Dewi(4), Rina(4), Sari(4)]
most visits  [Budi(9), Agus(7), Dewi(4), Rina(4), Sari(4)]
by city, then name Z-A:
  Gresik    Dewi(4)
  Gresik    Budi(9)
  Sidoarjo  Rina(4)
  Surabaya  Sari(4)
  Surabaya  Agus(7)
fewest visits Dewi(4)
one per count [Dewi(4), Agus(7), Budi(9)]
subtraction says low vs high = -4000000000 (Python ints never overflow)
```

Python's natural order is defined by `__lt__`; the decorator `@total_ordering` fills in `<=`, `>` and `>=` from it and `__eq__`. `sort(key=attrgetter("visits"), reverse=True)` sorts by visits, largest first, and Python's sort (Timsort) is stable too, so the ties keep name order [14]. For mixed directions on *strings* — city ascending, name descending — there is no number to negate. The Pythonic solution uses stability: sort by the *minor* key first (name, descending), then by the *major* key (city, ascending). The second sort keeps the first sort's order among customers of the same city. The dictionary built with `setdefault` plays the part of the `TreeSet` with a comparator — one customer per visit count, the first one seen — but makes the "one per count" rule visible instead of hiding it in a comparator. And Python's integers never overflow, so `low - high` is simply −4,000,000,000.

9.10 Choosing the Right Collection

We now know a dozen collections. Choosing among them is a matter of counting how much work each operation costs as the collection grows, in the O-notation used throughout this book: $O(1)$ means a bounded number of steps whatever the size, $O(\log n)$ a number of steps that grows by one each time n doubles, $O(n)$ a number proportional to n. **Table 9.5** gives the costs for Java [5], [9].

Table 9.5 Cost of the main operations of Java's collection classes for n elements.

Class	add / put	remove(x)	contains / get(key)	get(i)	Order when iterated
`ArrayList`	$O(1)$ at end (amortized); $O(n)$ elsewhere	$O(n)$	$O(n)$	$O(1)$	positions
`LinkedList`	$O(1)$ at ends; $O(n)$ to reach the middle	$O(n)$	$O(n)$	$O(n)$	positions
`ArrayDeque`	$O(1)$ at either end	$O(1)$ at ends	$O(n)$	—	front to back
`HashSet`, `HashMap`	$O(1)$ expected	$O(1)$ expected	$O(1)$ expected	—	unspecified
`LinkedHashSet`, `LinkedHashMap`	$O(1)$ expected	$O(1)$ expected	$O(1)$ expected	—	insertion
`TreeSet`, `TreeMap`	$O(\log n)$	$O(\log n)$	$O(\log n)$	—	sorted
`PriorityQueue`	$O(\log n)$	`poll`: $O(\log n)$; any other: $O(n)$	$O(n)$; `peek` $O(1)$	—	heap array

Table 9.6 gives the same for Python [7].

Table 9.6 Cost of the main operations of Python's collections for n elements.

Type	Add	Remove	`x in c` / `d[k]`	`c[i]`	Notes
`list`	`append` $O(1)$ amortized; `insert(0, x)` $O(n)$	`pop()` $O(1)$; `pop(0)`, `remove(x)` $O(n)$	$O(n)$	$O(1)$	Java's `ArrayList`
`tuple`	—	—	$O(n)$	$O(1)$	immutable, hashable if its elements are
`collections.deque`	$O(1)$ at either end	$O(1)$ at ends	$O(n)$	$O(n)$ in the middle	Java's `ArrayDeque`
`set`, `frozenset`	$O(1)$ expected	$O(1)$ expected	$O(1)$ expected	—	order varies for strings
`dict`	$O(1)$ expected	$O(1)$ expected	$O(1)$ expected	—	insertion order kept
`heapq` on a `list`	`heappush` $O(\log n)$	`heappop` $O(\log n)$	$O(n)$	`h[0]` is the minimum	Java's `PriorityQueue`
sorted `list` + `bisect`	`insort` $O(n)$ (search $O(\log n)$)	$O(n)$	$O(\log n)$ with `bisect`	$O(1)$	Java's `TreeSet` for read-mostly data

The difference between these classes is not academic. **Figure 9.9** plots the number of comparisons one `contains` makes, as the collection grows from ten to a million elements, on logarithmic axes.

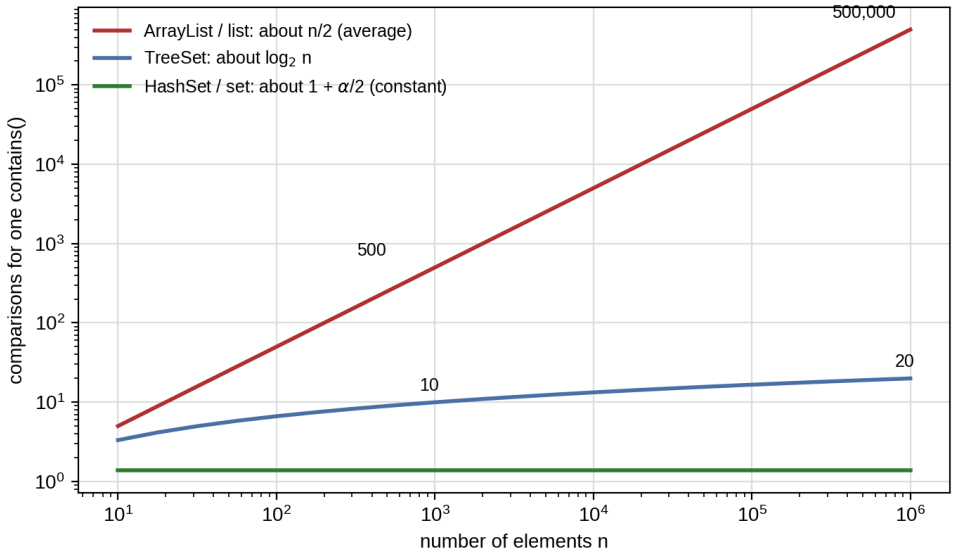


Figure 9.9 Comparisons made by one `contains` as n grows (both axes logarithmic): about $n/2$ for a list, about $\log_2 n$ for a balanced tree, and a small constant for a hash table with load factor $\alpha \leq 0.75$.

At a thousand elements a list needs about 500 comparisons, a tree about 10, a hash table one or two. At a million, the list needs half a million, the tree 20, the hash table still one or two. If the laundry's order book answered "is L-217 in the system?" with a list, each question on a million orders would cost as much as 25,000 questions answered by a tree set, or several hundred thousand answered by a hash set.

Numbers like these are the reason for the rule of thumb in **Figure 9.10**: ask four questions in order — look up by key? no duplicates? only the ends or smallest first? — and the answer names the collection; if all answers are "no", use a list.

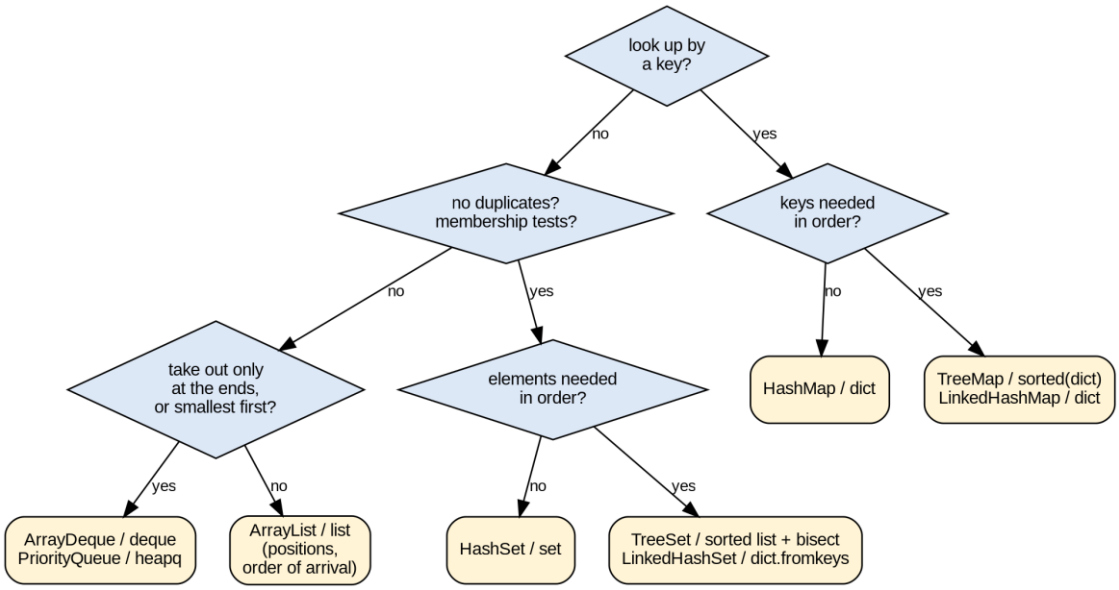


Figure 9.10 Choosing a collection with four questions. Each leaf names the Java class and the Python type that answer them.

Measure before optimizing: for a few dozen elements every collection is fast, and the clearest code wins. And when a program needs *several* of these answers for the same data — by number *and* by customer *and* by status — the solution is several collections that refer to the same objects, kept in agreement by one class. That is the design of the next section.

9.11 Profitina Laundry: an Order Book with Several Indexes

At the start of the chapter the owner asked four questions: *Where is order L-217? What has Bu Sari left with us? Which orders are ready? Which bag goes into the next free machine?* Each question now has a collection that answers it quickly:

- **find by order number**: a map from id to order, $O(\log n)$ with a `TreeMap`, which also lists the orders in id order;
- **orders of a customer**: a map from customer name to the list of that customer's orders;
- **orders in a status**: one set per status — `RECEIVED`, `WASHING`, `READY`, `PICKED_UP` — in an `EnumMap`, a map specialized for enum keys that stores its values in an array indexed by the constant's position;
- **the next wash**: a priority queue in which express orders come before normal ones and, within each group, the earlier arrival comes first.

Figure 9.11 shows the design with three of Sari's orders. The crucial point is that the four structures do not hold *copies* of the orders. They hold references to the *same* `LaundryOrder` objects, so an order changed through one index is changed for all.

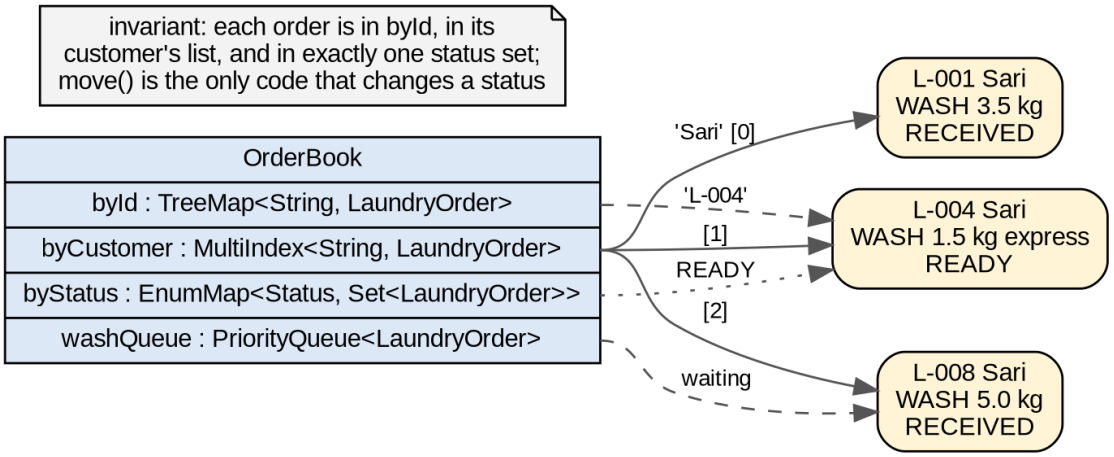


Figure 9.11 The order book: four structures, one set of `LaundryOrder` objects. The class keeps an invariant: every order appears in `byId`, in its customer's list, and in exactly one status set.

Several indexes bring a new duty: they must never disagree. If an order becomes `READY` but is still in the `WASHING` set, the answer to "which orders are ready?" is wrong, silently. The class therefore keeps an **invariant**, a statement that is true after every public method, and checks it in a method of its own:

$$\sum_{s \in Status} |byStatus[s]| = |byId| \quad \text{and} \quad o \in byStatus[o.status] \text{ for every order } o \tag{9.8}$$

The first half of Equation (9.8) says that the status sets together hold exactly as many orders as the book; the second that each order sits in the set of its own status. Together they say that every order is in exactly one status set. The design makes the invariant easy to keep: only one private method, `move`, ever changes an order's status, and it updates the index in the same breath.

The prices follow Chapter 5's rule: WASH Rp7,000, IRON Rp5,000 and WASH_IRON Rp10,000 per kilogram, and an express order costs half as much again. For weight w and rate r,

$$p = \operatorname{round}(w \cdot r) + e \cdot \left\lfloor \frac{\operatorname{round}(w \cdot r)}{2} \right\rfloor, \qquad e = \begin{cases} 1 & \text{express} \\ 0 & \text{otherwise} \end{cases} \tag{9.9}$$

so Sari's express 1.5 kg wash costs $10{,}500 + 5{,}250 = 15{,}750$. Iron-only orders need no machine: they never enter the wash queue and may go straight from `RECEIVED` to `READY`. **Listing 9.33** is the whole program.

Listing 9.33 `LaundryOrderBook.java`: Profitina Laundry's order book with a `TreeMap`, a generic `MultiIndex`, an `EnumMap` of `LinkedHashSet`s and a `PriorityQueue`, plus revenue reports sorted with comparators.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.EnumMap;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Set;
import java.util.TreeMap;

public class LaundryOrderBook {
    public static void main(String[] args) {
        OrderBook book = new OrderBook();
        book.receive(new LaundryOrder("L-001", "Sari", "WASH", 3.5, false));
        book.receive(new LaundryOrder("L-002", "Budi", "WASH_IRON", 5.0, true));
        book.receive(new LaundryOrder("L-003", "Rina", "IRON", 2.0, false));
        book.receive(new LaundryOrder("L-004", "Sari", "WASH", 1.5, true));
        book.receive(new LaundryOrder("L-005", "Agus", "WASH", 6.0, false));
        book.receive(new LaundryOrder("L-006", "Budi", "WASH", 4.0, false));
        book.receive(new LaundryOrder("L-007", "Dewi", "WASH_IRON", 2.5, true));
        book.receive(new LaundryOrder("L-008", "Sari", "WASH", 5.0, false));
        System.out.println("wash queue: " + book.washQueueInOrder());

        for (int i = 0; i < 3; i++) {
            System.out.println("start wash " + book.startNextWash().getId());
        }
        book.markReady("L-002");
        book.markReady("L-004");
        book.markReady("L-003");                // iron only: no wash needed
        book.pickUp("L-002");
        try {
            book.pickUp("L-005");
        } catch (IllegalStateException e) {
            System.out.println("refused: " + e.getMessage());
        }
        try {
            book.receive(new LaundryOrder("L-001", "Joko", "IRON", 1.0, false));
        } catch (IllegalArgumentException e) {
            System.out.println("refused: " + e.getMessage());
        }
```

Listing 9.33 (Continued) *LaundryOrderBook.java: Profitina Laundry's order book with a TreeMap, a generic MultiIndex, an EnumMap of LinkedHashSets and a PriorityQueue, plus revenue reports sorted with comparators.*

```
        System.out.println("status counts " + book.statusCounts());
        System.out.println("Sari's orders " + book.ordersOf("Sari"));
        System.out.println("READY        " + book.withStatus(Status.READY));
        System.out.println("heaviest 3   " + book.heaviest(3));
        System.out.println("invariant holds: " + book.invariantHolds());
        book.printRevenueReport();
    }
}

enum Status { RECEIVED, WASHING, READY, PICKED_UP }

final class LaundryOrder {
    private static final Map<String, Integer> RATES = Map.of(
            "WASH", 7000, "IRON", 5000, "WASH_IRON", 10000);
    private static int arrivals = 0;          // gives every order an arrival number

    private final String id;
    private final String customer;
    private final String service;
    private final double weightKg;
    private final boolean express;
    private final int arrival;
    private Status status = Status.RECEIVED;  // the only field that changes

    LaundryOrder(String id, String customer, String service, double weightKg, boolean
express) {
        if (!RATES.containsKey(service)) {
            throw new IllegalArgumentException("unknown service " + service);
        }
        this.id = id;
        this.customer = customer;
        this.service = service;
        this.weightKg = weightKg;
        this.express = express;
        this.arrival = ++arrivals;
    }

    String getId() { return id; }
    String getCustomer() { return customer; }
    String getService() { return service; }
    double getWeightKg() { return weightKg; }
    boolean isExpress() { return express; }
    int getArrival() { return arrival; }
    Status getStatus() { return status; }
    void setStatus(Status s) { status = s; }   // package-private: only OrderBook calls it
```

Listing 9.33 (Continued) *LaundryOrderBook.java: Profitina Laundry's order book with a TreeMap, a generic MultiIndex, an EnumMap of LinkedHashSets and a PriorityQueue, plus revenue reports sorted with comparators.*

```
    boolean needsWash() {
        return !service.equals("IRON");
    }

    long price() {                          // Chapter 5's rule: express adds half
        long base = Math.round(weightKg * RATES.get(service));
        return express ? base + base / 2 : base;
    }

    @Override
    public boolean equals(Object o) {       // identity of an order = its id
        return o instanceof LaundryOrder other && id.equals(other.id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }

    @Override
    public String toString() {
        return id;
    }
}

// A generic index: one key, many values, keys kept in sorted order.
final class MultiIndex<K extends Comparable<K>, V> {
    private final Map<K, List<V>> map = new TreeMap<>();

    void add(K key, V value) {
        map.computeIfAbsent(key, k -> new ArrayList<>()).add(value);
    }

    List<V> get(K key) {
        return Collections.unmodifiableList(map.getOrDefault(key, List.of()));
    }

    Set<K> keys() {
        return Collections.unmodifiableSet(map.keySet());
    }
}
```

Listing 9.33 (Continued) *LaundryOrderBook.java: Profitina Laundry's order book with a TreeMap, a generic MultiIndex, an EnumMap of LinkedHashSets and a PriorityQueue, plus revenue reports sorted with comparators.*

```
final class OrderBook {
    private final Map<String, LaundryOrder> byId = new TreeMap<>();
    private final MultiIndex<String, LaundryOrder> byCustomer = new MultiIndex<>();
    private final Map<Status, Set<LaundryOrder>> byStatus = new EnumMap<>(Status.class);
    // Express first; within each group, first come, first served.
    private final PriorityQueue<LaundryOrder> washQueue = new PriorityQueue<>(
            Comparator.comparing((LaundryOrder o) -> !o.isExpress())
                      .thenComparingInt(LaundryOrder::getArrival));

    OrderBook() {
        for (Status s : Status.values()) {
            byStatus.put(s, new LinkedHashSet<>());
        }
    }

    void receive(LaundryOrder order) {
        if (byId.containsKey(order.getId())) {
            throw new IllegalArgumentException("duplicate id " + order.getId());
        }
        byId.put(order.getId(), order);
        byCustomer.add(order.getCustomer(), order);
        byStatus.get(Status.RECEIVED).add(order);
        if (order.needsWash()) {
            washQueue.offer(order);
        }
    }

    LaundryOrder startNextWash() {
        LaundryOrder next = washQueue.poll();
        if (next == null) {
            throw new IllegalStateException("no order is waiting to be washed");
        }
        move(next, Status.WASHING);
        return next;
    }

    void markReady(String id) {
        LaundryOrder o = find(id);
        boolean allowed = o.getStatus() == Status.WASHING
                || (o.getStatus() == Status.RECEIVED && !o.needsWash());
        if (!allowed) {
            throw new IllegalStateException(id + " cannot become READY from " +
o.getStatus());
        }
        move(o, Status.READY);
    }
```

Listing 9.33 (Continued) *LaundryOrderBook.java: Profitina Laundry's order book with a TreeMap, a generic MultiIndex, an EnumMap of LinkedHashSets and a PriorityQueue, plus revenue reports sorted with comparators.*

```
    void pickUp(String id) {
        LaundryOrder o = find(id);
        if (o.getStatus() != Status.READY) {
            throw new IllegalStateException(id + " is " + o.getStatus() + ", not READY");
        }
        move(o, Status.PICKED_UP);
    }

    // The one place where the status index changes: remove, update, add back.
    private void move(LaundryOrder o, Status to) {
        byStatus.get(o.getStatus()).remove(o);
        o.setStatus(to);
        byStatus.get(to).add(o);
    }

    private LaundryOrder find(String id) {
        LaundryOrder o = byId.get(id);
        if (o == null) {
            throw new IllegalArgumentException("no order " + id);
        }
        return o;
    }

    List<LaundryOrder> ordersOf(String customer) {
        return byCustomer.get(customer);
    }

    List<LaundryOrder> withStatus(Status s) {
        return new ArrayList<>(byStatus.get(s));
    }

    Map<Status, Integer> statusCounts() {
        Map<Status, Integer> counts = new EnumMap<>(Status.class);
        for (Map.Entry<Status, Set<LaundryOrder>> e : byStatus.entrySet()) {
            counts.put(e.getKey(), e.getValue().size());
        }
        return counts;
    }
```

Listing 9.33 (Continued) *LaundryOrderBook.java: Profitina Laundry's order book with a TreeMap, a generic MultiIndex, an EnumMap of LinkedHashSets and a PriorityQueue, plus revenue reports sorted with comparators.*

```
    List<LaundryOrder> washQueueInOrder() {
        PriorityQueue<LaundryOrder> copy = new PriorityQueue<>(washQueue);
        List<LaundryOrder> result = new ArrayList<>();
        while (!copy.isEmpty()) {
            result.add(copy.poll());
        }
        return result;
    }

    List<LaundryOrder> heaviest(int k) {
        List<LaundryOrder> all = new ArrayList<>(byId.values());
        all.sort(Comparator.comparingDouble(LaundryOrder::getWeightKg).reversed()
                .thenComparing(LaundryOrder::getId));
        return all.subList(0, Math.min(k, all.size()));
    }

    // Every order sits in exactly one status set.
    boolean invariantHolds() {
        int total = 0;
        for (Set<LaundryOrder> set : byStatus.values()) {
            total += set.size();
        }
        for (LaundryOrder o : byId.values()) {
            if (!byStatus.get(o.getStatus()).contains(o)) {
                return false;
            }
        }
        return total == byId.size();
    }

    void printRevenueReport() {
        Map<String, Long> perCustomer = new TreeMap<>();
        Map<String, long[]> perService = new TreeMap<>();    // {orders, rupiah}
        long total = 0;
        for (LaundryOrder o : byId.values()) {
            perCustomer.merge(o.getCustomer(), o.price(), Long::sum);
            long[] cell = perService.computeIfAbsent(o.getService(), k -> new long[2]);
            cell[0]++;
            cell[1] += o.price();
            total += o.price();
        }
        List<Map.Entry<String, Long>> ranking = new ArrayList<>(perCustomer.entrySet());
        ranking.sort(Map.Entry.<String, Long>comparingByValue().reversed()
                .thenComparing(Map.Entry.comparingByKey()));
```

Listing 9.33 (Continued) *LaundryOrderBook.java: Profitina Laundry's order book with a TreeMap, a generic MultiIndex, an EnumMap of LinkedHashSets and a PriorityQueue, plus revenue reports sorted with comparators.*

```
        System.out.printf("%-22s %6s %12s%n", "REVENUE BY CUSTOMER", "orders", "amount");
        for (Map.Entry<String, Long> e : ranking) {
            System.out.printf("  %-20s %6d %12s%n", e.getKey(),
                    ordersOf(e.getKey()).size(), rupiah(e.getValue()));
        }
        System.out.printf("%-22s %6s %12s%n", "REVENUE BY SERVICE", "orders", "amount");
        for (Map.Entry<String, long[]> e : perService.entrySet()) {
            System.out.printf("  %-20s %6d %12s%n", e.getKey(),
                    e.getValue()[0], rupiah(e.getValue()[1]));
        }
        System.out.printf("  %-20s %6d %12s%n", "TOTAL", byId.size(), rupiah(total));
    }

    static String rupiah(long amount) {
        return String.format(Locale.US, "Rp%,d", amount);
    }
}
```

```
wash queue: [L-002, L-004, L-007, L-001, L-005, L-006, L-008]
start wash L-002
start wash L-004
start wash L-007
refused: L-005 is RECEIVED, not READY
refused: duplicate id L-001
status counts {RECEIVED=4, WASHING=1, READY=2, PICKED_UP=1}
Sari's orders [L-001, L-004, L-008]
READY        [L-004, L-003]
heaviest 3   [L-005, L-002, L-008]
invariant holds: true
REVENUE BY CUSTOMER    orders       amount
  Budi                      2    Rp103,000
  Sari                      3     Rp75,250
  Agus                      1     Rp42,000
  Dewi                      1     Rp37,500
  Rina                      1     Rp10,000
REVENUE BY SERVICE     orders       amount
  IRON                      1     Rp10,000
  WASH                      5    Rp145,250
  WASH_IRON                 2    Rp112,500
  TOTAL                     8    Rp267,750
```

Let us read it class by class.

`Status` is an enum with the four stages of an order's life. Using an enum rather than `int` constants (Section 9.2.1) means that `byStatus` can be an `EnumMap<Status, ...>` and that a status can never be 7.

`LaundryOrder` holds an order's data. All fields are `final` except `status`, whose setter is package-private. The static counter `arrivals` numbers the orders; the priority queue uses it to break ties. `equals` and `hashCode` use the id alone, and that is essential: if the hash depended on the status, changing the status would strand the order in the wrong bucket of its `LinkedHashSet` (Exercise 9.8). `RATES` is a `Map.of(...)`, an *immutable* map: `put` on it throws `UnsupportedOperationException`.

`MultiIndex<K extends Comparable<K>, V>` is a small generic class of our own: one key, many values, keys kept in a `TreeMap` (hence the bound). `add` uses `computeIfAbsent(key, k -> new ArrayList<>())`, which returns the key's list, creating it the first time the key is seen. `get` returns `Collections.unmodifiableList(...)`, a read-only *view*, so nobody can break the invariant

through the list we hand out; an absent key gives the empty `List.of()` rather than `null`. Being generic, the class knows nothing about laundry and could index books by title unchanged.

`OrderBook` owns the four structures, all `private final`. The constructor puts an empty `LinkedHashSet` into the `EnumMap` for every status, so that `byStatus.get(s)` is never `null`. The queue's comparator compares `!o.isExpress()`; since `false` sorts before `true`, express orders come first, and `.thenComparingInt(LaundryOrder::getArrival)` breaks ties by arrival.

- `receive` refuses a duplicate id with an exception *before* touching any structure, so a refused order leaves no trace. Then it adds the order to every index — and to the wash queue if it needs washing.
- `startNextWash` polls the queue and moves the order to `WASHING`.
- `markReady` and `pickUp` check that the transition is allowed — an order can become ready only after washing, or directly if it is iron-only; it can be picked up only when ready — and throw `IllegalStateException` otherwise.
- `move` is the single place where a status changes: remove from the old set, change the field, add to the new set. Removal must happen *before* the field changes, while the order can still be found in its old set.
- `washQueueInOrder` polls a *copy* of the queue; printing the `PriorityQueue` itself would show the heap array, not the order of service.
- `printRevenueReport` sums prices per customer with `merge` and per service with `computeIfAbsent`, then ranks customers by `Map.Entry.<String, Long>comparingByValue().reversed().thenComparing(Map.Entry.comparingByKey())`; the explicit `<String, Long>` is needed because `.reversed()` is applied before the compiler could infer the types.

Now match the output with `main`. The wash queue holds the three express washes first, `L-002, L-004, L-007`, then the four normal ones; iron-only `L-003` is not in it. Three washes start; `L-002` and `L-004` finish, `L-003` is ironed, and `L-002` is picked up. The book refuses `L-005`, still `RECEIVED`, and a second `L-001`. The status counts, in the enum's order, add up to $4 + 1 + 2 + 1 = 8$, as Equation (9.8) demands, and `invariant holds` checks it explicitly. `READY` lists `L-004` before `L-003`, the order in which they became ready, thanks to `LinkedHashSet`.

The revenue report can be checked by hand with Equation (9.9). Budi: round$(5.0 \times 10{,}000) = 50{,}000$, plus half for express, 75,000; and $4.0 \times 7{,}000 = 28{,}000$; together 103,000. Sari: $3.5 \times 7{,}000 = 24{,}500$, the express 15,750, and $5.0 \times 7{,}000 = 35{,}000$; together 75,250. Agus 42,000, Dewi $25{,}000 + 12{,}500 = 37{,}500$, Rina $2.0 \times 5{,}000 = 10{,}000$. The total is 267,750, and the per-service lines add up to the same: $10{,}000 + 145{,}250 + 112{,}500 = 267{,}750$.

Listing 9.34 is the Python version; its output is the same except for Python's spelling of dictionaries and booleans.

Listing 9.34 `laundry_order_book.py`: the same order book in Python, with a generic `MultiIndex`, dictionaries used as ordered sets, and `heapq` for the wash queue.

```
"""laundry_order_book.py: Profitina Laundry's order book, indexed three ways."""
from __future__ import annotations

import heapq
import itertools
from enum import Enum
from typing import Any, Generic, Protocol, TypeVar


class Status(Enum):
    RECEIVED = 1
    WASHING = 2
    READY = 3
    PICKED_UP = 4


class LaundryOrder:
    RATES = {"WASH": 7000, "IRON": 5000, "WASH_IRON": 10000}
    _arrivals = itertools.count(1)            # gives every order an arrival number

    def __init__(self, order_id: str, customer: str, service: str,
                 weight_kg: float, express: bool) -> None:
        if service not in self.RATES:
            raise ValueError(f"unknown service {service}")
        self.id = order_id
        self.customer = customer
        self.service = service
        self.weight_kg = weight_kg
        self.express = express
        self.arrival = next(self._arrivals)
        self.status = Status.RECEIVED         # the only attribute that changes

    def needs_wash(self) -> bool:
        return self.service != "IRON"

    def price(self) -> int:                   # Chapter 5's rule: express adds half
        base = round(self.weight_kg * self.RATES[self.service])
        return base + base // 2 if self.express else base

    def __eq__(self, other: object) -> bool:  # identity of an order = its id
        return isinstance(other, LaundryOrder) and self.id == other.id

    def __hash__(self) -> int:
        return hash(self.id)
```

Listing 9.34 (Continued) *laundry_order_book.py: the same order book in Python, with a generic MultiIndex, dictionaries used as ordered sets, and heapq for the wash queue.*

```
    def __repr__(self) -> str:
        return self.id


class SupportsLessThan(Protocol):
    def __lt__(self, other: Any, /) -> bool: ...


K = TypeVar("K", bound=SupportsLessThan)
V = TypeVar("V")


class MultiIndex(Generic[K, V]):
    """A generic index: one key, many values, keys reported in sorted order."""

    def __init__(self) -> None:
        self._map: dict[K, list[V]] = {}

    def add(self, key: K, value: V) -> None:
        self._map.setdefault(key, []).append(value)

    def get(self, key: K) -> tuple[V, ...]:    # a tuple: the caller cannot change it
        return tuple(self._map.get(key, ()))

    def keys(self) -> list[K]:
        return sorted(self._map)


class OrderBook:
    def __init__(self) -> None:
        self._by_id: dict[str, LaundryOrder] = {}
        self._by_customer: MultiIndex[str, LaundryOrder] = MultiIndex()
        # dict.fromkeys-style sets that remember insertion order: a dict with None values
        self._by_status: dict[Status, dict[LaundryOrder, None]] = {s: {} for s in Status}
        # Express first; within each group, first come, first served.
        self._wash_queue: list[tuple[bool, int, LaundryOrder]] = []

    def receive(self, order: LaundryOrder) -> None:
        if order.id in self._by_id:
            raise ValueError(f"duplicate id {order.id}")
        self._by_id[order.id] = order
        self._by_customer.add(order.customer, order)
        self._by_status[Status.RECEIVED][order] = None
        if order.needs_wash():
            heapq.heappush(self._wash_queue, (not order.express, order.arrival, order))
```

Listing 9.34 (Continued) *laundry_order_book.py: the same order book in Python, with a generic MultiIndex, dictionaries used as ordered sets, and heapq for the wash queue.*

```
    def start_next_wash(self) -> LaundryOrder:
        if not self._wash_queue:
            raise RuntimeError("no order is waiting to be washed")
        _, _, order = heapq.heappop(self._wash_queue)
        self._move(order, Status.WASHING)
        return order

    def mark_ready(self, order_id: str) -> None:
        o = self._find(order_id)
        allowed = o.status is Status.WASHING or (
            o.status is Status.RECEIVED and not o.needs_wash())
        if not allowed:
            raise RuntimeError(f"{order_id} cannot become READY from {o.status.name}")
        self._move(o, Status.READY)

    def pick_up(self, order_id: str) -> None:
        o = self._find(order_id)
        if o.status is not Status.READY:
            raise RuntimeError(f"{order_id} is {o.status.name}, not READY")
        self._move(o, Status.PICKED_UP)

    def _move(self, o: LaundryOrder, to: Status) -> None:
        """The one place where the status index changes: remove, update, add back."""
        del self._by_status[o.status][o]
        o.status = to
        self._by_status[to][o] = None

    def _find(self, order_id: str) -> LaundryOrder:
        try:
            return self._by_id[order_id]
        except KeyError:
            raise ValueError(f"no order {order_id}") from None

    def orders_of(self, customer: str) -> tuple[LaundryOrder, ...]:
        return self._by_customer.get(customer)

    def with_status(self, s: Status) -> list[LaundryOrder]:
        return list(self._by_status[s])

    def status_counts(self) -> dict[str, int]:
        return {s.name: len(orders) for s, orders in self._by_status.items()}

    def wash_queue_in_order(self) -> list[LaundryOrder]:
        return [entry[2] for entry in sorted(self._wash_queue)]
```

Listing 9.34 (Continued) *laundry_order_book.py: the same order book in Python, with a generic MultiIndex, dictionaries used as ordered sets, and heapq for the wash queue.*

```
    def heaviest(self, k: int) -> list[LaundryOrder]:
        return sorted(self._by_id.values(), key=lambda o: (-o.weight_kg, o.id))[:k]

    def invariant_holds(self) -> bool:
        """Every order sits in exactly one status set."""
        total = sum(len(orders) for orders in self._by_status.values())
        in_right_set = all(o in self._by_status[o.status] for o in self._by_id.values())
        return in_right_set and total == len(self._by_id)

    def print_revenue_report(self) -> None:
        per_customer: dict[str, int] = {}
        per_service: dict[str, list[int]] = {}            # [orders, rupiah]
        for o in self._by_id.values():
            per_customer[o.customer] = per_customer.get(o.customer, 0) + o.price()
            cell = per_service.setdefault(o.service, [0, 0])
            cell[0] += 1
            cell[1] += o.price()
        ranking = sorted(per_customer.items(), key=lambda kv: (-kv[1], kv[0]))

        print(f"{'REVENUE BY CUSTOMER':<22} {'orders':>6} {'amount':>12}")
        for name, amount in ranking:
            print(f"  {name:<20} {len(self.orders_of(name)):6d} {rupiah(amount):>12}")
        print(f"{'REVENUE BY SERVICE':<22} {'orders':>6} {'amount':>12}")
        for service in sorted(per_service):
            count, amount = per_service[service]
            print(f"  {service:<20} {count:6d} {rupiah(amount):>12}")
        total = sum(per_customer.values())
        print(f"  {'TOTAL':<20} {len(self._by_id):6d} {rupiah(total):>12}")


def rupiah(amount: int) -> str:
    return f"Rp{amount:,}"


def main() -> None:
    book = OrderBook()
    book.receive(LaundryOrder("L-001", "Sari", "WASH", 3.5, False))
    book.receive(LaundryOrder("L-002", "Budi", "WASH_IRON", 5.0, True))
    book.receive(LaundryOrder("L-003", "Rina", "IRON", 2.0, False))
    book.receive(LaundryOrder("L-004", "Sari", "WASH", 1.5, True))
    book.receive(LaundryOrder("L-005", "Agus", "WASH", 6.0, False))
    book.receive(LaundryOrder("L-006", "Budi", "WASH", 4.0, False))
    book.receive(LaundryOrder("L-007", "Dewi", "WASH_IRON", 2.5, True))
    book.receive(LaundryOrder("L-008", "Sari", "WASH", 5.0, False))
    print("wash queue:", book.wash_queue_in_order())
```

Listing 9.34 (Continued) *laundry_order_book.py: the same order book in Python, with a generic MultiIndex, dictionaries used as ordered sets, and heapq for the wash queue.*

```
    for _ in range(3):
        print("start wash", book.start_next_wash().id)
    book.mark_ready("L-002")
    book.mark_ready("L-004")
    book.mark_ready("L-003")                  # iron only: no wash needed
    book.pick_up("L-002")
    try:
        book.pick_up("L-005")
    except RuntimeError as e:
        print("refused:", e)
    try:
        book.receive(LaundryOrder("L-001", "Joko", "IRON", 1.0, False))
    except ValueError as e:
        print("refused:", e)

    print("status counts", book.status_counts())
    print("Sari's orders", list(book.orders_of("Sari")))
    print(f"READY        {book.with_status(Status.READY)}")
    print(f"heaviest 3   {book.heaviest(3)}")
    print("invariant holds:", book.invariant_holds())
    book.print_revenue_report()


if __name__ == "__main__":
    main()
```

```
wash queue: [L-002, L-004, L-007, L-001, L-005, L-006, L-008]
start wash L-002
start wash L-004
start wash L-007
refused: L-005 is RECEIVED, not READY
refused: duplicate id L-001
status counts {'RECEIVED': 4, 'WASHING': 1, 'READY': 2, 'PICKED_UP': 1}
Sari's orders [L-001, L-004, L-008]
READY        [L-004, L-003]
heaviest 3   [L-005, L-002, L-008]
invariant holds: True
REVENUE BY CUSTOMER    orders       amount
  Budi                      2    Rp103,000
  Sari                      3     Rp75,250
  Agus                      1     Rp42,000
  Dewi                      1     Rp37,500
  Rina                      1     Rp10,000
REVENUE BY SERVICE     orders       amount
  IRON                      1     Rp10,000
  WASH                      5    Rp145,250
  WASH_IRON                 2    Rp112,500
  TOTAL                     8    Rp267,750
```

The structure is class for class the same; the interesting differences are in how each Java collection is spelled.

- **Ordered sets.** Python has no `LinkedHashSet`, but a `dict` keeps insertion order, so a dictionary whose values are all `None` is an ordered set: `self._by_status[s][order] = None` adds, `del self._by_status[s][order]` removes, and `list(...)` gives the elements in the order they arrived. The dictionary comprehension `{s: {} for s in Status}` creates one per status.
- **The wash queue.** `heapq` compares the pushed items with `<`, and a `LaundryOrder` has no `<`. So each entry is a tuple `(not order.express, order.arrival, order)`: tuples compare element by element, `False` sorts before `True` so express orders come first, and the arrival numbers are

unique, so the comparison never reaches the order object itself. This tuple trick is the standard way to give `heapq` a custom priority.

- **The generic index.** `MultiIndex(Generic[K, V])`, with `K` bounded by the same `SupportsLessThan` protocol as in **Listing 9.24**, uses `setdefault(key, [])`, the counterpart of `computeIfAbsent`, and returns a *tuple* from `get`, so the caller receives an immutable snapshot rather than the index's own list.
- **Arrival numbers.** `itertools.count(1)` is an endless iterator 1, 2, 3, …, shared by all orders as a class attribute, like Java's `static` field. mypy checks the whole file without a complaint.

9.12 Profitina in Practice

Profitina is an AI visibility platform: it scans a brand's website and measures how ready the site is to be quoted by AI answer engines, the discipline called *generative engine optimization* (GEO). The product is written in Python, and the collections of this chapter are its everyday tools, used in exactly the roles the order book gave them.

A scan visits many pages of a site, and a crawl must never fetch the same page twice. Before a URL is queued it is brought to a *canonical* form — one spelling for the many ways of writing the same address — and a **set** of canonical URLs answers "seen it already?" in constant expected time, the question of Equation (9.5). The pages waiting to be fetched form a **queue**; when some pages matter more than others, it becomes a **priority queue**, the structure of Section 9.7. The results are kept in **maps** from page to measurements and from signal to score, and the weekly trend a brand owner sees on the dashboard is a map from week to score, read in sorted order and exported as CSV. The community feed keeps "N new" counters per channel: a map from channel to count, updated with the equivalent of `merge`.

In a code base of that size, a type hint such as `dict[str, list[Signal]]` states in one line what a function returns, which is the bargain of **Table 9.4**; the release gate itself is the automated suite of well over a thousand tests, which must pass before a version ships. And, as the order book showed, several indexes over the same objects stay correct only when one class owns them and one method changes them. As always, the measurements are advice: the platform explains what it found, and the user decides what to do.

9.13 Common Mistakes, and How to Avoid Them

1. Declaring the class instead of the interface. `ArrayList<String> names = new ArrayList<>();` ties every method that receives `names` to one implementation. *Avoid it:* declare variables and parameters with the interface, `List<String>`, `Map<String, Order>`; name the class only after `new` (Section 9.3.2).

2. Removing from a collection inside a for-each loop. Java throws `ConcurrentModificationException`; Python silently skips elements (**Listing 9.8**). *Avoid it:* use `Iterator.remove`, `removeIf`, a comprehension, or a loop over a copy.

3. Calling `get(i)` in a loop over a `LinkedList`, or `pop(0)` in a loop over a Python `list`. Each call is $O(n)$, so the loop is $O(n^2)$. *Avoid it:* walk a linked list with for-each; use `ArrayDeque` or `collections.deque` for a queue.

4. Using `list.contains` or `x in list` for frequent membership tests. Each test scans the list. *Avoid it:* keep a `HashSet` or `set` beside the list, or instead of it (**Figure 9.9**).

5. Relying on the iteration order of a `HashSet`, `HashMap` or Python `set`. The order depends on hash codes and table size, and for Python strings it changes on every run. *Avoid it:* choose `LinkedHashSet`/`LinkedHashMap` (insertion order) or `TreeSet`/`TreeMap` (sorted), or sort explicitly when printing.

6. Changing a field that `hashCode` or `compareTo` depends on while the object is in a hash set, a map key, or a tree. The object is stranded in the wrong bucket or the wrong branch; `contains` and `remove` no longer find it (Exercise 9.8). *Avoid it:* base `equals`/`hashCode` on fields that never change, such as an id, and make them `final`; in Python, give `__hash__` only to objects whose hashed fields are immutable.

7. Writing a comparator with subtraction. `a - b` overflows for large or negative values and reports the wrong order (**Listing 9.31**). *Avoid it:* `Integer.compare(a, b)`, `Comparator.comparingInt(...)`; in Python, a `key=` function.

8. Using a comparator that calls different objects "equal" in a `TreeSet` or `TreeMap`. The tree keeps only one of them. *Avoid it:* add tie-breakers until "compares equal" means "is the same element".

9. Using raw types or ignoring "unchecked" warnings. They switch off the compiler's checks and allow heap pollution (**Listing 9.27**). *Avoid it:* always write type arguments, compile with `-Xlint:all`, and fix every warning; suppress one only with a comment that explains why it is safe.

10. Expecting `List<Cat>` to be a `List<Animal>`. It is not (Equation (9.7)). *Avoid it:* a method that only reads should take `List<? extends Animal>` (Python: `Sequence[Animal]`); a method that only adds should take `List<? super Cat>`.

11. Believing that Python type hints are checked when the program runs. They are not; `list[int]` accepts strings at run time (**Listing 9.28**). *Avoid it:* run mypy (or another checker) as part of testing, and annotate the functions you want checked [3], [15].

12. `d[key]` on a missing key in Python, `map.get(key).method()` on a missing key in Java. The first raises `KeyError`, the second `NullPointerException`. *Avoid it:* `d.get(key, default)`, `getOrDefault`, `computeIfAbsent`, `defaultdict`.

13. Letting several indexes over the same objects drift apart. An order in the wrong status set gives wrong answers silently. *Avoid it:* one class owns all the indexes, only one method changes the indexed field, and an invariant check such as Equation (9.8) runs in the tests.

9.14 Chapter Summary

- An interface may declare **constants**; they are automatically `public static final`. For a fixed set of related values, an **enum** is safer in both Java and Python. Python's `Final` is checked by mypy, not at run time.
- A **recursive** method needs a base case that every chain of calls reaches and a recursive call on a smaller problem. The naive Fibonacci makes $2F(n+1)-1$ calls (Equation (9.2)); carrying state in extra parameters makes it linear, and a **helper method** hides those parameters from callers.
- The **Collections Framework** (**Figure 9.2**) names the kinds of collection with interfaces — `Collection`, `List`, `Set`, `Queue`, `Deque`, and the separate `Map` — and implements each in several ways. Python's counterparts are the built-in `list`, `tuple`, `set`, `frozenset`, `dict`, plus `deque`, `Counter`, `defaultdict` and `heapq` (**Table 9.2**). Program to the interface.

- **Lists** keep positions and duplicates. `ArrayList` / `list` give $O(1)$ access by index and cheap appends; inserting at the front costs $O(n)$ (Equation (9.3)). Walk with for-each or an `Iterator`; remove through the iterator, `removeIf` or a comprehension.
- **Sets** keep no duplicates. A **hash set** computes an element's bucket from its hash code and answers `contains` in expected $O(1)$ time, because the load factor (Equation (9.4)) is kept below 0.75 (Equation (9.5)); a **tree set** keeps sorted order in a red-black tree of height $O(\log n)$ (Equation (9.6)); a **linked hash set** keeps insertion order.
- **Maps** associate unique keys with values: `put` returns the old value, `get` returns `null` (Python: `KeyError`, or `get` with a default), `merge`/`Counter` count, `computeIfAbsent`/`defaultdict` group.
- **Queues** are FIFO, **stacks** LIFO, both served by `ArrayDeque` / `deque`; a **priority queue** (`PriorityQueue`, `heapq`) is a binary heap that releases the smallest element in $O(\log n)$.
- **Generics** let one class or method work for any type while the compiler checks every use: type parameters `<T>`, generic methods `<T> T f(...)`, bounds `<T extends Comparable<T>>`. Generic types are **invariant** (Equation (9.7)); the wildcards `? extends` (producers, read) and `? super` (consumers, write) restore flexibility — PECS (**Table 9.3**). Java **erases** type arguments at run time; Python never had them at run time, and mypy checks its `TypeVar`, `Generic` and `list[int]` hints (**Table 9.4**).
- Order comes from `Comparable` (natural order) or a `Comparator` (any order); build comparators with `comparing`, `thenComparing` and `reversed`; Java's object sort and Python's sort are **stable**. Python sorts by **key functions**.
- **Choose a collection** by the questions you will ask (**Figure 9.10**) and the costs in **Table 9.5** and **Table 9.6**. The laundry's order book combines a `TreeMap`, a generic `MultiIndex`, an `EnumMap` of sets and a `PriorityQueue` over the same objects, kept consistent by one invariant (Equation (9.8)).

9.15 Exercises

Exercise 9.1 (choosing). For each task, name the Java collection class and the Python type you would use, and justify the choice in one sentence using **Figure 9.10** and **Table 9.5**. (a) The tickets printed today, in the order printed. (b) The set of customers who have visited this month, asked "has she visited?" hundreds of times a day. (c) The price of each service code. (d) Orders waiting for the single ironing table, served in order of arrival. (e) The customers' names in alphabetical order, with the question "who comes right after Rina?". (f) Delivery jobs for the courier, always taking the one with the earliest promised time. (g) An undo history in which the most recent action is undone first.

Exercise 9.2 (tracing). Without running it, predict the single line printed by

```
List<Integer> a = new ArrayList<>(List.of(4, 1, 4, 2));
Set<Integer> s = new TreeSet<>(a);
Map<Integer, Integer> m = new TreeMap<>();
for (int x : a) {
    m.merge(x, 1, Integer::sum);
}
a.remove(Integer.valueOf(4));
a.remove(1);
Deque<Integer> d = new ArrayDeque<>(a);
d.push(9);
d.offerLast(7);
System.out.println(a + " " + s + " " + m + " " + d + " " + d.pollLast() + d.pop());
```

Then write the same steps in Python with `list`, `set`, `Counter` and `deque`, and predict its output.

Exercise 9.3 (recursion with a helper). Add to `GList<T>` of **Listing 9.19** a method `GList<T> reversed()` that returns a new list with the elements in the opposite order. Write it recursively, with a private helper that carries an accumulator, so that it makes $O(n)$ calls. Do the same in Python. Why is the "obvious" version — reverse the tail, then append the head at the end — slower for a list built from `cons` cells?

Exercise 9.4 (anagrams). Two words are *anagrams* if they use the same letters the same number of times: "listen", "silent", "enlist" and "tinsel". Write a program that groups the words `one, listen, neo, silent, loop, pool, polo, enlist, eon, aab, abb, tinsel` into anagram groups and prints every group with more than one member. Use a map whose key is the word's letters in sorted order. Explain why comparing *sets* of letters, as in **Listing 9.9**, would be wrong.

Exercise 9.5 (a generic stack). Write a generic class `ArrayStack<T>` in Java, and `ArrayStack(Generic[T])` in Python, with `push`, `pop`, `peek`, `isEmpty`/`is_empty` and `size`/`__len__`, backed by a private list whose *end* is the top. `pop` and `peek` on an empty stack must throw a clear exception. Use the stack in a function `balanced(text)` that checks whether the brackets `()`, `[]` and `{}` in a text are properly matched and nested, and test it on `(a[b]{c})`, `(a[b)]`, `((` and `x}`.

Exercise 9.6 (`Comparable<? super T>`). Given

```
class Fruit implements Comparable<Fruit> {
    final int grams;
    Fruit(int grams) { this.grams = grams; }
    public int compareTo(Fruit other) { return Integer.compare(grams, other.grams); }
}
class Apple extends Fruit {
    Apple(int grams) { super(grams); }
}
```

explain why `max(apples)` fails to compile for `List<Apple> apples` when `max` is declared as in **Listing 9.23**, `static <T extends Comparable<T>> T max(List<T> items)`. Change the bound so that it compiles, and explain the change with PECS. (This is why the library declares `Collections.max` as `<T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)`.)

Exercise 9.7 (wildcards). With `Cat` and `Dog` extending `Animal`, and the declarations `List<Cat> cats`, `List<Animal> animals`, `List<Object> objects`, `List<? extends Animal> readers`, `List<? super Cat> writers`, say which of these statements compile, and why: (a) `readers = cats;` (b) `readers = objects;` (c) `writers = animals;` (d) `writers = cats;` (e) `Animal a = readers.get(0);` (f) `readers.add(new Cat());` (g) `writers.add(new Cat());` (h) `Cat c = writers.get(0);` (i) `Object o = writers.get(0);` (j) `animals = cats;`.

Exercise 9.8 (a hash code that changes). A class `Tag` has a mutable field `code`, and its `equals` and `hashCode` use `code`. Predict the two lines printed by

```
Set<Tag> set = new HashSet<>();
Tag t = new Tag("WASH");
set.add(t);
t.code = "IRON";
System.out.println(set.contains(t) + " " + set.contains(new Tag("WASH")) + " " +
set.size());
set.add(t);
System.out.println(set.size() + " " + set);
```

(`toString` returns `code`.) Explain each value with **Figure 9.5**. What does the same experiment do in Python, and what is the lesson for `LaundryOrder` in **Listing 9.33**?

Exercise 9.9 (counting the cost). (a) Using Equation (9.3), how many element moves does inserting $n = 100{,}000$ elements one by one at index 0 of an `ArrayList` cause? How many link changes does the same job cost with a `LinkedList` or an `ArrayDeque`'s `addFirst`? (b) A list of 10^6 customer ids contains duplicates. Compare the number of comparisons made by removing duplicates with a nested loop ("for each element, is it among the elements kept so far?") and with a `HashSet`, assuming at most 10^6 distinct ids. (c) How many levels does a binary heap of 10^6 elements have, and so at most how many swaps does one `poll` make? (d) By Equation (9.6), at most how many comparisons does `contains` make on a `TreeSet` of 10^6 elements?

Exercise 9.10 (an LRU cache). A *least recently used* (LRU) cache holds at most c entries; when a new entry would exceed the capacity, it evicts the entry that has gone longest without being read or written. Implement a generic `LruCache<K, V>` in Java by extending `LinkedHashMap` with *access order* and overriding `removeEldestEntry`, and a generic `LruCache(Generic[K, V])` in Python with `collections.OrderedDict` and `move_to_end`. Demonstrate with capacity 3: put `L-001`, `L-002`, `L-003`, get `L-001`, put `L-004`, and show which entry was evicted. Chapter 7 warned against extending concrete classes; why is extending `LinkedHashMap` acceptable here?

Exercise 9.11 (cancelling an order). Add `cancel(id)` to the order book of **Listing 9.33** and **Listing 9.34**. Only an order that is still `RECEIVED` may be cancelled; it must disappear from *every* structure, so that the invariant Equation (9.8) still holds and Budi's list no longer shows it. You will need a `remove(key, value)` method in `MultiIndex`. What does removing an arbitrary element cost in a `PriorityQueue` and in a `heapq` list, and what alternative ("lazy deletion") avoids that cost?

Exercise 9.12 (variance in Python). mypy rejects the call in the file `weights.py`

```
class Order:
    def __init__(self, weight: float) -> None:
        self.weight = weight


class ExpressOrder(Order):
    pass


def total_weight(orders: list[Order]) -> float:
    return sum(o.weight for o in orders)


def main() -> None:
    express: list[ExpressOrder] = [ExpressOrder(2.5), ExpressOrder(4.0)]
    print(total_weight(express))


main()
```

although the program runs and prints `6.5`. Explain mypy's objection with Equation (9.7), fix the signature so that mypy accepts the call, and give an example of a function body for which mypy's objection would be justified.

9.16 Solutions to the Exercises

Solution 9.1. (a) `ArrayList` / `list`: positions and order of arrival, appends at the end. (b) `HashSet` / `set`: membership tested often, $O(1)$ expected. (c) `HashMap` / `dict` (or an `EnumMap` if the codes are an enum): look-up by key. (d) `ArrayDeque` used as a queue / `collections.deque`: FIFO, both ends $O(1)$. (e) `TreeSet` / a sorted `list` with `bisect`: sorted order and `higher("Rina")`, the

smallest element strictly greater; in Python `names[bisect_right(names, "Rina")]`. (f) `PriorityQueue` with a comparator on the promised time / `heapq` with tuples `(time, seq, job)`. (g) `ArrayDeque` used as a stack (`push`, `pop`) / `list` with `append` and `pop`. (Java's old class `Stack` also works but extends `Vector` and is not recommended.)

Solution 9.2. The Java program prints

```
[1, 2] [1, 2, 4] {1=1, 2=1, 4=2} [9, 1, 2, 7] 79
```

`s` is the sorted set of distinct values; `m` counts them (4 occurs twice). `a.remove(Integer.valueOf(4))` removes the *value* 4, its first occurrence, leaving `[1, 4, 2]`; `a.remove(1)` removes *index* 1 (Chapter 5's boxing trap), leaving `[1, 2]`. The deque is built from `[1, 2]`; `push` adds 9 at the front, `offerLast` 7 at the back. String concatenation is evaluated left to right, so `d` is printed before `pollLast()` returns 7 and `pop()` returns 9; `7 + 9` does not add, because the left operand is already a `String`, so the characters `79` are appended. The Python version

```
a = [4, 1, 4, 2]; s = sorted(set(a)); m = dict(sorted(Counter(a).items()))
a.remove(4); del a[1]
d = deque(a); d.appendleft(9); d.append(7)
print(a, s, m, list(d), d.pop(), d.popleft())
```

prints `[1, 2] [1, 2, 4] {1: 1, 2: 1, 4: 2} [9, 1, 2, 7] 7 9`. `list(d)` makes a snapshot before the two pops run, because arguments are evaluated left to right before `print` starts.

Solution 9.3. In Java:

```
public GList<T> reversed() {
    return reversedHelper(this, new GList<>());
}

private GList<T> reversedHelper(GList<T> rest, GList<T> accumulator) {
    if (rest.empty()) {
        return accumulator;                                    // base case
    }
    return reversedHelper(rest.tail(), new GList<>(rest.head(), accumulator));
}
```

Each call moves one element from the front of `rest` to the front of `accumulator`, like moving plates from one pile to another; after n calls the accumulator holds the elements reversed. In Python, `def reversed(self) -> GList[T]: return self._rev(self, GList.nil())` with `_rev(rest, acc)` returning `acc` if `rest.empty()` and `self._rev(rest.tail(), GList.cons(rest.head(), acc))` otherwise. The "obvious" version must append at the *end* of the reversed tail, and a `cons` list can only add at the front; appending means copying the whole list, $O(k)$ for a list of k elements, so the total is $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$, the same sum as Equation (9.3): $O(n^2)$.

Solution 9.4. **Listing 9.35** and **Listing 9.36** group the words by their sorted letters.

Listing 9.35 `Anagrams.java`: anagram groups in a `TreeMap` keyed by the sorted letters of each word.

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;

public class Anagrams {
    // "neo", "one" and "eon" all have the key "eno": their letters, sorted.
    static String key(String word) {
        char[] letters = word.toLowerCase().toCharArray();
        Arrays.sort(letters);
        return new String(letters);
    }

    static Map<String, List<String>> groups(List<String> words) {
        Map<String, List<String>> result = new TreeMap<>();
        for (String w : words) {
            result.computeIfAbsent(key(w), k -> new ArrayList<>()).add(w);
        }
        return result;
    }

    public static void main(String[] args) {
        List<String> words = List.of("one", "listen", "neo", "silent", "loop",
                "pool", "polo", "enlist", "eon", "aab", "abb", "tinsel");
        Map<String, List<String>> g = groups(words);
        for (Map.Entry<String, List<String>> e : g.entrySet()) {
            if (e.getValue().size() > 1) {
                System.out.println(e.getKey() + " -> " + e.getValue());
            }
        }
        System.out.println(g.size() + " groups from " + words.size() + " words");
    }
}
```

```
eilnst -> [listen, silent, enlist, tinsel]
eno -> [one, neo, eon]
loop -> [loop, pool, polo]
5 groups from 12 words
```

Listing 9.36 `anagrams.py`: the same grouping with a `defaultdict`.

```python
"""anagrams.py: group words that are anagrams of each other."""
from collections import defaultdict


def key(word: str) -> str:
    """'neo', 'one' and 'eon' all have the key 'eno': their letters, sorted."""
    return "".join(sorted(word.lower()))


def groups(words: list[str]) -> dict[str, list[str]]:
    result: defaultdict[str, list[str]] = defaultdict(list)
    for w in words:
        result[key(w)].append(w)
    return dict(sorted(result.items()))


def main() -> None:
    words = ["one", "listen", "neo", "silent", "loop", "pool", "polo", "enlist",
             "eon", "aab", "abb", "tinsel"]
    g = groups(words)
    for k, members in g.items():
        if len(members) > 1:
            print(f"{k} -> {members}")
    print(f"{len(g)} groups from {len(words)} words")
    print("set('aab') == set('abb'):", set("aab") == set("abb"), "- but they are not
anagrams")


if __name__ == "__main__":
    main()
```

```
eilnst -> ['listen', 'silent', 'enlist', 'tinsel']
eno -> ['one', 'neo', 'eon']
loop -> ['loop', 'pool', 'polo']
5 groups from 12 words
set('aab') == set('abb'): True - but they are not anagrams
```

`key("listen")` sorts the letters to `eilnst`; every anagram of “listen” has the same key, and no other word does, because the sorted letters record each letter *with its multiplicity*. `computeIfAbsent` and `defaultdict(list)` create each group on first use. Twelve words form five groups: the three printed, and the singletons `aab` and `abb`. A set of letters forgets how often each letter occurs, so `set("aab") == set("abb")`, as the last Python line shows, although the words are not anagrams.

Solution 9.5. **Listing 9.37** and **Listing 9.38** implement the stack and the bracket checker.

Listing 9.37 `StackDemo.java`: a generic `ArrayStack<T>` whose top is the end of a private `ArrayList`, and a bracket checker built on it.

```
import java.util.ArrayList;
import java.util.List;
import java.util.NoSuchElementException;

public class StackDemo {
    // Is every bracket closed by the right partner, in the right order?
    static boolean balanced(String text) {
        ArrayStack<Character> open = new ArrayStack<>();
        for (char c : text.toCharArray()) {
            if (c == '(' || c == '[' || c == '{') {
                open.push(c);
            } else if (c == ')' || c == ']' || c == '}') {
                if (open.isEmpty()) {
                    return false;
                }
                char o = open.pop();
                if ((c == ')' && o != '(') || (c == ']' && o != '[') || (c == '}' && o !=
'{')) {
                    return false;
                }
            }
        }
        return open.isEmpty();
    }

    public static void main(String[] args) {
        ArrayStack<String> trays = new ArrayStack<>();
        trays.push("tray A");
        trays.push("tray B");
        trays.push("tray C");
        System.out.println(trays + " size " + trays.size() + ", peek " + trays.peek());
        System.out.println("pop " + trays.pop() + ", pop " + trays.pop() + ", left " +
trays);

        for (String s : new String[] {"(a[b]{c})", "(a[b)]", "((", "x}"}) {
            System.out.println(s + " balanced: " + balanced(s));
        }
        try {
            new ArrayStack<Integer>().pop();
        } catch (NoSuchElementException e) {
            System.out.println("NoSuchElementException: " + e.getMessage());
        }
    }
}
```

Listing 9.37 (Continued) *StackDemo.java: a generic ArrayStack<T> whose top is the end of a private ArrayList, and a bracket checker built on it.*

```
// A generic stack. The ArrayList is private: callers see only stack operations.
class ArrayStack<T> {
    private final List<T> items = new ArrayList<>();

    void push(T item) {
        items.add(item);                    // the top is the END of the list: O(1)
    }

    T pop() {
        T top = peek();
        items.remove(items.size() - 1);
        return top;
    }

    T peek() {
        if (items.isEmpty()) {
            throw new NoSuchElementException("stack is empty");
        }
        return items.get(items.size() - 1);
    }

    boolean isEmpty() {
        return items.isEmpty();
    }

    int size() {
        return items.size();
    }

    @Override
    public String toString() {
        return "bottom " + items + " top";
    }
}
```

```
bottom [tray A, tray B, tray C] top size 3, peek tray C
pop tray C, pop tray B, left bottom [tray A] top
(a[b]{c}) balanced: true
(a[b)] balanced: false
(( balanced: false
x} balanced: false
NoSuchElementException: stack is empty
```

Listing 9.38 `stack_demo.py`: the generic stack in Python, and the same bracket checker with a partner table.

```
"""stack_demo.py: a generic stack class, and a bracket checker that uses it."""
from typing import Generic, TypeVar

T = TypeVar("T")


class ArrayStack(Generic[T]):
    """A generic stack. The list is private: callers see only stack operations."""

    def __init__(self) -> None:
        self._items: list[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)            # the top is the END of the list: O(1)

    def pop(self) -> T:
        top = self.peek()
        self._items.pop()
        return top

    def peek(self) -> T:
        if not self._items:
            raise IndexError("stack is empty")
        return self._items[-1]

    def is_empty(self) -> bool:
        return not self._items

    def __len__(self) -> int:
        return len(self._items)

    def __str__(self) -> str:
        return f"bottom {self._items} top"


PARTNER = {")": "(", "]": "[", "}": "{"}
```

Listing 9.38 (Continued) *stack_demo.py: the generic stack in Python, and the same bracket checker with a partner table.*

```
def balanced(text: str) -> bool:
    """Is every bracket closed by the right partner, in the right order?"""
    open_brackets: ArrayStack[str] = ArrayStack()
    for c in text:
        if c in "([{":
            open_brackets.push(c)
        elif c in PARTNER:
            if open_brackets.is_empty() or open_brackets.pop() != PARTNER[c]:
                return False
    return open_brackets.is_empty()


def main() -> None:
    trays: ArrayStack[str] = ArrayStack()
    trays.push("tray A")
    trays.push("tray B")
    trays.push("tray C")
    print(f"{trays} size {len(trays)}, peek {trays.peek()}")
    print(f"pop {trays.pop()}, pop {trays.pop()}, left {trays}")

    for s in ["(a[b]{c})", "(a[b)]", "((", "x}"]:
        print(f"{s} balanced: {balanced(s)}")
    try:
        ArrayStack[int]().pop()
    except IndexError as e:
        print("IndexError:", e)


if __name__ == "__main__":
    main()
```

```
bottom ['tray A', 'tray B', 'tray C'] top size 3, peek tray C
pop tray C, pop tray B, left bottom ['tray A'] top
(a[b]{c}) balanced: True
(a[b)] balanced: False
(( balanced: False
x} balanced: False
IndexError: stack is empty
```

The top is the *end* of the list, so `push` and `pop` are $O(1)$, and the list is private, so no caller can insert in the middle: composition, not inheritance (Chapter 7). The checker pushes every opening bracket; a closing bracket must match the most recently opened one, which is exactly the element on top. `(a[b)]` fails because `)` arrives while `[` is on top; `((` fails because two brackets remain open at the end; `x}` fails because the stack is empty when `}` arrives.

Solution 9.6. For `List<Apple>`, the compiler must choose `T = Apple` (the list's element type) and then check the bound `Apple extends Comparable<Apple>`. It fails: `Apple` inherits `Comparable<Fruit>`, not `Comparable<Apple>`, and generic types are invariant. In a test program `SuperBound.java` in which the strict version is named `maxStrict`, `javac` says:

```
SuperBound.java:19: error: method maxStrict in class SuperBound cannot be applied to given
types;
        System.out.println(maxStrict(apples));
                           ^
  required: List<T>
  found:    List<Apple>
  reason: inference variable T has incompatible equality constraints Fruit,Apple
  where T is a type-variable:
    T extends Comparable<T> declared in method <T>maxStrict(List<T>)
1 error
```

The fix is `static <T extends Comparable<? super T>> T max(List<T> items)`: `T` must be comparable with *some supertype* of itself. `Apple` is comparable with `Fruit`, a supertype of `Apple`, so the bound holds, and `max(apples)` returns the 210 g apple. In PECS terms, the `Comparable` *consumes* `T` values (its `compareTo` takes one as an argument), so its type argument should be `? super T`.

Solution 9.7. (a) Compiles: `List<Cat>` is a subtype of `List<? extends Animal>`. (b) Does not: `Object` is not a subtype of `Animal`. (c) Compiles: `Animal` is a supertype of `Cat`. (d) Compiles: `Cat` is a supertype of itself. (e) Compiles: whatever `readers` holds, its elements are `Animal`s. (f) Does not: `readers` might be a `List<Dog>`. (g) Compiles: whatever `writers` holds can hold a `Cat`. (h) Does not: `writers` might be a `List<Object>`, so `get` returns only an `Object`; a cast would compile but could fail. (i) Compiles. (j) Does not: Equation (9.7).

Solution 9.8. The program prints

```
false false 1
2 [IRON, IRON]
```

When `t` was added, its hash code was that of `"WASH"`, so it was stored in `"WASH"`'s bucket. After `t.code = "IRON"`, `contains(t)` computes the bucket of `"IRON"`, looks there, and finds nothing: `false`. `contains(new Tag("WASH"))` looks in the right bucket and finds `t`, but `equals` compares `"WASH"` with `t`'s current code `"IRON"`: `false`. The size is still 1. `set.add(t)` then looks in `"IRON"`'s bucket, finds nothing, and adds `t` *a second time*: the "set" now holds the same object twice. Python's `set` behaves identically (`False False 1`, then `2`). The lesson: fields used by `equals`/`hashCode` (and by `compareTo` in a tree) must not change while the object is in a collection. That is why `LaundryOrder` bases both on the `final` id, and why its mutable `status` plays no part in them.

Solution 9.9. (a) $\frac{n(n-1)}{2} = \frac{100{,}000 \times 99{,}999}{2} = 4{,}999{,}950{,}000$ moves, about five billion. `LinkedList.addFirst` changes three references per insertion, about $3n = 300{,}000$ in all; `ArrayDeque.addFirst` writes one slot per insertion, about n writes plus occasional doubling. (b) The nested loop compares each of the 10^6 elements with up to 10^6 kept elements, up to about $\frac{10^{12}}{2}$ comparisons in the worst case; a hash set makes about 10^6 expected $O(1)$ steps: a factor of roughly half a million. (c) $\lfloor \log_2 10^6 \rfloor + 1 = 19 + 1 = 20$ levels, so one `poll` makes at most 19 swaps as the moved element sinks. (d) $2\log_2(10^6 + 1) \approx 2 \times 19.93 \approx 39.9$, so at most 39 levels below the root, 40 comparisons.

Solution 9.10. **Listing 9.39** and **Listing 9.40** are the two caches.

Listing 9.39 `LruCache.java`: an LRU cache in a dozen lines, by extending `LinkedHashMap` in access order.

```java
import java.util.LinkedHashMap;
import java.util.Map;

// A cache that forgets the Least Recently Used entry when it is full.
public class LruCache<K, V> extends LinkedHashMap<K, V> {
    private static final long serialVersionUID = 1L;
    private final int capacity;

    public LruCache(int capacity) {
        super(16, 0.75f, true);            // true = order by ACCESS, not by insertion
        this.capacity = capacity;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > capacity;          // called by put: evict when over capacity
    }

    public static void main(String[] args) {
        LruCache<String, String> receipts = new LruCache<>(3);
        receipts.put("L-001", "Sari 24,500");
        receipts.put("L-002", "Budi 75,000");
        receipts.put("L-003", "Rina 10,000");
        System.out.println(receipts.keySet());
        receipts.get("L-001");             // L-001 becomes the most recently used
        System.out.println("after get L-001  " + receipts.keySet());
        receipts.put("L-004", "Sari 15,750");
        System.out.println("after put L-004  " + receipts.keySet() + " (L-002 evicted)");
        System.out.println("get L-002 -> " + receipts.get("L-002"));
    }
}
```

```
[L-001, L-002, L-003]
after get L-001  [L-002, L-003, L-001]
after put L-004  [L-003, L-001, L-004] (L-002 evicted)
get L-002 -> null
```

Listing 9.40 `lru_cache.py`: the same cache with `OrderedDict` and `move_to_end`.

```python
"""lru_cache.py: a cache that forgets the Least Recently Used entry when it is full."""
from collections import OrderedDict
from typing import Generic, Hashable, Optional, TypeVar

K = TypeVar("K", bound=Hashable)
V = TypeVar("V")


class LruCache(Generic[K, V]):
    def __init__(self, capacity: int) -> None:
        self._capacity = capacity
        self._data: OrderedDict[K, V] = OrderedDict()

    def get(self, key: K) -> Optional[V]:
        if key not in self._data:
            return None
        self._data.move_to_end(key)          # now the most recently used
        return self._data[key]

    def put(self, key: K, value: V) -> None:
        self._data[key] = value
        self._data.move_to_end(key)
        if len(self._data) > self._capacity:
            self._data.popitem(last=False)   # evict the least recently used

    def keys(self) -> list[K]:
        return list(self._data)


def main() -> None:
    receipts: LruCache[str, str] = LruCache(3)
    receipts.put("L-001", "Sari 24,500")
    receipts.put("L-002", "Budi 75,000")
    receipts.put("L-003", "Rina 10,000")
    print(receipts.keys())
    receipts.get("L-001")
    print("after get L-001 ", receipts.keys())
    receipts.put("L-004", "Sari 15,750")
    print("after put L-004 ", receipts.keys(), "(L-002 evicted)")
    print("get L-002 ->", receipts.get("L-002"))


if __name__ == "__main__":
    main()
```

```
['L-001', 'L-002', 'L-003']
after get L-001  ['L-002', 'L-003', 'L-001']
after put L-004  ['L-003', 'L-001', 'L-004'] (L-002 evicted)
get L-002 -> None
```

The constructor `LinkedHashMap(16, 0.75f, true)` sets the initial capacity, the load factor of Equation (9.4) and, with `true`, *access order*: every `get` or `put` moves the entry to the end of the internal linked list, so the head is always the least recently used. After every `put`, `LinkedHashMap` calls `removeEldestEntry(head)`, and our override answers “yes, remove it” when the size exceeds the capacity. The trace: after three puts the order is `L-001, L-002, L-003`; `get("L-001")` moves it to the end; `put("L-004")` makes the size 4, so the head `L-002` is evicted. Extending a concrete class is acceptable here because `LinkedHashMap` was *designed* for it: `removeEldestEntry` is a documented, `protected` hook whose only purpose is to be overridden, and the documentation shows exactly this use [5]. The fragile base class problem of Chapter 7 arises when a subclass depends on undocumented internals. In Python, `OrderedDict.move_to_end(key)` marks an entry most recent and `popitem(last=False)` removes the oldest. (For a *function*, `functools.lru_cache` of **Listing 9.4** does all of this for you.)

Solution 9.11. Add to `MultiIndex` in Java

```
    void remove(K key, V value) {
        List<V> list = map.get(key);
        if (list != null && list.remove(value) && list.isEmpty()) {
            map.remove(key);                  // no empty lists left behind
        }
    }
```

and to `OrderBook`

```
    void cancel(String id) {
        LaundryOrder o = find(id);
        if (o.getStatus() != Status.RECEIVED) {
            throw new IllegalStateException(id + " is " + o.getStatus() + " and cannot be
    cancelled");
        }
        washQueue.remove(o);                  // O(n): a heap cannot find o faster
        byStatus.get(Status.RECEIVED).remove(o);
        byCustomer.remove(o.getCustomer(), o);
        byId.remove(id);
    }
```

In Python, `MultiIndex.remove` does the same with `values.remove(value)` and `del self._map[key]`, and `cancel` rebuilds the heap without the order: `self._wash_queue = [e for e in self._wash_queue if e[2] is not o]`, then `heapq.heapify(self._wash_queue)`, followed by the three deletions. Adding `book.cancel("L-006")` and an attempt to cancel `L-004` after the refusals in `main` prints, in both languages,

```
refused: L-004 is READY and cannot be cancelled
after cancel: queue [L-001, L-005, L-008], Budi [L-002]
```

and the invariant still holds, with `RECEIVED` down to 3. `PriorityQueue.remove(o)` must first *find* `o`, and a heap is ordered only along each path from the root, so the search is linear, $O(n)$, followed by an $O(\log n)$ repair; `heapify` is $O(n)$ as well. *Lazy deletion* avoids the search: keep a set of cancelled ids, leave the entry in the heap, and when `startNextWash` polls an entry whose id is in the set, discard it and poll again. Each cancelled entry is then removed exactly once, at $O(\log n)$, when it reaches the top.

Solution 9.12. mypy reports

```
weights.py:16: error: Argument 1 to "total_weight" has incompatible type
"list[ExpressOrder]"; expected "list[Order]"  [arg-type]
weights.py:16: note: "list" is invariant -- see
https://mypy.readthedocs.io/en/stable/common_issues.html#variance
weights.py:16: note: Consider using "Sequence" instead, which is covariant
Found 1 error in 1 file (checked 1 source file)
```

`ExpressOrder` is a subtype of `Order`, but by Equation (9.7) `list[ExpressOrder]` is not a subtype of `list[Order]`. The function only reads, so declare `def total_weight(orders: Sequence[Order]) -> float` (from `collections.abc` or `typing`); `Sequence` is covariant and mypy accepts the call. The objection would be justified for a body such as `orders.append(Order(1.0))`: with `list[Order]` it type-checks inside the function, and if a `list[ExpressOrder]` were accepted, a plain `Order` would end up in a list that promises only express orders — the dog among the cats of **Listing 9.26**.

References

[1] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*. Oracle America, 2023. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se21/html/

[2] J. Bloch, *Effective Java*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2018.

[3] The mypy project, *mypy documentation*: "Generics" and "Common issues and solutions: variance". [Online]. Available: https://mypy.readthedocs.io/

[4] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed. Hoboken, NJ, USA: Wiley, 2014.

[5] Oracle, *Java Platform, Standard Edition 21 API Specification*, package `java.util`, and "The Collections Framework" overview. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/

[6] S. Marks, "JEP 431: Sequenced Collections," OpenJDK, 2023 (delivered in JDK 21). [Online]. Available: https://openjdk.org/jeps/431

[7] Python Software Foundation, *The Python Standard Library*, version 3.14: `collections`, `collections.abc`, `heapq`, `bisect`, `typing`; and "Time complexity" of built-in operations. [Online]. Available: https://docs.python.org/3/library/

[8] L. Ramalho, *Fluent Python*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2022.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[10] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. 19th Annu. Symp. Foundations of Computer Science (FOCS)*, Ann Arbor, MI, USA, 1978, pp. 8–21.

[11] G. Bracha, "Generics in the Java programming language," Sun Microsystems tutorial, 2004. [Online]. Available: https://www.oracle.com/technetwork/java/javase/generics-tutorial-159168.pdf

[12] E. Traut, "PEP 695 – Type parameter syntax," Python Enhancement Proposal 695, 2022. [Online]. Available: https://peps.python.org/pep-0695/

[13] M. Naftalin and P. Wadler, *Java Generics and Collections*. Sebastopol, CA, USA: O'Reilly Media, 2006.

[14] T. Peters, "listsort.txt: description of Timsort," CPython source distribution, 2002. [Online]. Available: https://github.com/python/cpython/blob/main/Objects/listsort.txt

[15] G. van Rossum, J. Lehtosalo, and Ł. Langa, "PEP 484 – Type hints," Python Enhancement Proposal 484, 2014. [Online]. Available: https://peps.python.org/pep-0484/

Chapter 10

Graphical User Interfaces and Event Handling

By the end of this chapter you will be able to build a real window in Java with Swing and in Python with Tkinter; explain the layers underneath (AWT, Swing, Tcl/Tk); arrange components with layout managers and with `pack` and `grid`; draw your own shapes in `paintComponent` and on a `Canvas`; explain from nothing what an *event loop* is; make a program respond to clicks, keys, check boxes and mouse movements with *listeners* and *callbacks*; write those listeners as inner classes, anonymous classes, lambdas and method references in Java, and as bound methods, closures, lambdas and `partial` objects in Python, knowing what each one captures; obey the one rule every GUI toolkit imposes on threads; and test a window on a machine with no screen. Everything comes together in a complete order-entry window for Profitina Laundry, written in Swing and in Tkinter, with a form, validation, a live price, a list of orders and a total, and with a self-test that proves the two versions behave identically.

10.1 Where We Left Off

Every program in this book so far has had the same shape. It starts at the top of `main`, does its work in the order the programmer wrote, prints some lines, and stops. Chapter 5 let programs read input, but even then the program was in charge: it printed a prompt, *waited* for the user to type a line, and carried on. Chapter 7 gave Profitina Laundry a family of service classes whose prices are computed polymorphically, and Chapter 9 stored many objects in collections and made those collections type-safe with generics.

The staff at the laundry counter, however, do not use a terminal. They use a window: a form with boxes to type the customer's name and the weight of the bag, a drop-down list of services, a tick box for express, a button that says *Add order*, and a list that shows the day's orders with a total at the bottom. They click wherever they like, in whatever order they like. They may type the weight before the name. They may change the service three times before adding the order. They may remove an order they added by mistake. The program cannot print a prompt and wait for one particular answer, because it has no idea what the user will do next.

That single fact — *the user, not the program, decides what happens next* — changes how a program is organized, and it is the subject of this chapter. We will meet the new vocabulary (component, container, layout, event, listener, callback, event loop), learn two real toolkits that implement it (Swing for Java, Tkinter for Python), and discover along the way three language features that make event-driven code pleasant to write: inner classes, anonymous classes and lambdas in Java, and closures in Python. These features are useful far beyond windows; you will meet them again in Chapter 13 (iterators) and Chapter 14 (threads).

> ✓ **KEY INSIGHT.** A console program *calls* the user ("type the weight now"). A GUI program *is called by* the user: it sets up a window, says "when this button is clicked, run that method", and then hands control to a loop that waits for things to happen. Everything in this chapter follows from that reversal.

10.2 From the Console to the Window: Who Is in Charge?

Let us make the reversal concrete. Here is how a console version of the laundry's order taking would run, written as the conversation it has with the user:

```
Customer name: Siti Aminah
Phone: 0812-3456-7890
Service (WF/IR/WDI): WF
Weight in kg: 4,2
Express (y/n): n
Added ORD-0001, Rp31,500. Another order (y/n)?
```

The program asks five questions in a fixed order. If Siti changes her mind about express after the weight has been typed, there is no way back: the question has already been asked. If the weight is mistyped, the program must either start the whole order again or contain special code to re-ask that one question. The order of the conversation is written into the code, as the order of the statements.

Now picture the same task as a window, which **Figure 10.18** in Section 10.12 shows exactly as it appears on the screen. All five questions are visible at once, each in its own box. The user fills them in any order, corrects any of them at any time, and sees the price change the moment the weight or the service changes. Only when the user clicks *Add order* does the program check the whole form and either accept the order or explain, in red, what is wrong.

The program that drives such a window does not contain the conversation at all. Instead it contains many small pieces of code, each answering one question of the form "what should happen when *this* occurs?":

- when the text in the weight box changes, recompute the price label;
- when a different service is chosen, recompute the price label;
- when the express box is ticked or unticked, recompute the price label;
- when *Add order* is clicked, validate the form, and either add the order or show an error;
- when a row of the list is selected, enable the *Remove selected* button;
- when *Remove selected* is clicked, remove that order and update the total.

Each of these occurrences is an **event**, and each small piece of code is an **event handler**. The main program only builds the window, connects each handler to the event it handles, and then gives up control to the toolkit, which runs forever (until the window is closed), waiting for events and calling the right handler for each. This style of program is called **event-driven**, and the reversal it embodies — the library calls your code, rather than your code calling the library — is known as **inversion of control** [1].

> **DEFINITION.** A **graphical user interface** (GUI) presents a program's input and output as visual elements — windows, buttons, text boxes, lists, drawings — that the user operates with a mouse, a keyboard or a touch screen. An **event** is anything that happens that the program may want to respond to: a click, a key press, a mouse movement, a window being resized or uncovered, a timer firing. An **event handler** (called a **listener** in Java and a **callback** in Python) is a method or function that the toolkit calls when a particular event occurs. A program organized as a set of handlers waiting for events is **event-driven**.

Three consequences follow at once, and we will meet each of them properly later in the chapter.

1. **Handlers must be short.** While one handler runs, no other event can be handled: the window cannot even redraw itself. A handler that takes three seconds freezes the window for three seconds. Section 10.11 shows how to move slow work elsewhere.

2. **State lives in objects, not in the position in the code.** A console program knows that "the weight has been typed" because execution has passed the line that reads it. A GUI program knows it only because some object remembers it — the text of a text field, or a field of your own class. That is why GUIs and objects go so well together, and why every window in this chapter is a class.
3. **The window must be described, not narrated.** Instead of a sequence of prompts, the program declares a tree of components and how they are arranged. Sections 10.5 and 10.8 are about that description.

10.3 Three Toolkits: AWT, Swing, and Tkinter

A **GUI toolkit** is a library of ready-made visual building blocks — windows, buttons, text fields, lists — together with the machinery that delivers events to them and draws them on the screen. You could, in principle, draw every button pixel by pixel yourself, but nobody does: a toolkit gives you a button that already knows how to look pressed, how to show keyboard focus, and how to tell you that it was clicked. This chapter uses the two toolkits that ship with the languages themselves, so that every program runs on a fresh installation with nothing extra to download. **Figure 10.1** shows how the pieces stack.

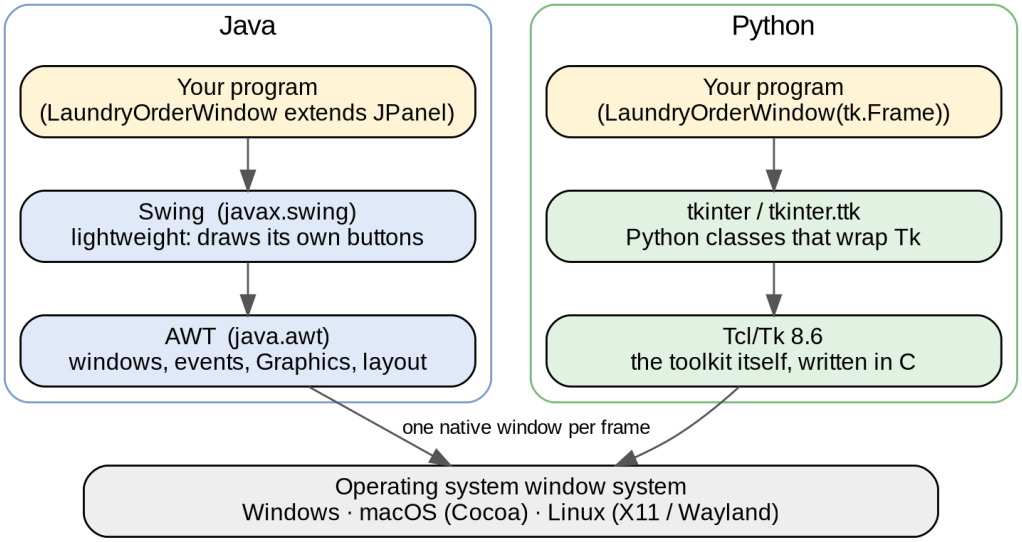


Figure 10.1 The layers under a window. In Java, your panel is built from Swing components, which rest on AWT, which asks the operating system for a native window. In Python, `tkinter` is a thin layer of Python classes over the Tcl/Tk toolkit.

10.3.1 AWT: the original toolkit

Java's first GUI library is the **Abstract Window Toolkit**, the package `java.awt`, which dates from JDK 1.0. In AWT every button was a real button of the operating system, a *native peer*: a Windows button on Windows, a Mac button on a Mac. That looked natural but hurt portability. Different systems offer different widgets with different behaviors, so AWT could offer only what all systems had in common, and programs still behaved slightly differently from one system to another. Components that rely on native resources this way are called **heavyweight**.

AWT is not obsolete, however. It still provides the foundations that everything else stands on: the window that the operating system actually opens, the classes that describe events (`ActionEvent`, `MouseEvent`, …) and the listener interfaces that receive them (package `java.awt.event`), the

`Graphics` class used for drawing, colors and fonts, and the layout managers. That is why almost every Swing program begins by importing from both `java.awt` and `javax.swing`.

10.3.2 Swing: components that draw themselves

In late 1996 Sun Microsystems began the **Java Foundation Classes** (JFC), whose GUI part was released with Java 2 (JDK 1.2) as the package `javax.swing`. *Swing* is not an acronym; it was the project's code name, and it stuck. Swing's components are **lightweight**: a `JButton` is not a native button at all. Swing asks the operating system for one native window per frame and then *paints* every button, label and list inside it itself, with Java code and the `Graphics` class. The result looks and behaves the same everywhere, and its appearance can even be changed by switching the *look and feel*, at the price of a little speed that nobody notices on today's machines [1].

Because a program usually imports both `java.awt` and `javax.swing`, and both have a class for a button, a label, a frame and so on, Swing prefixes its class names with the letter **J**: `JFrame`, `JButton`, `JLabel`, `JTextField`, `JPanel`, `JMenuBar`. Without the J, `Button` would be ambiguous. A rule of thumb that will save you hours: **in a Swing program, use only the J classes for components**; mixing an AWT `Button` into a Swing window causes drawing problems that are hard to diagnose.

In Java's vocabulary a top-level window with a title bar and borders is a **frame**, class `JFrame`. The ancestors of the classes you will use are shown on the left of **Figure 10.2**. Two things are worth noticing in it. First, `JFrame` descends from AWT's `Window` and `Frame`, because it *is* a native window; every other Swing component descends from `JComponent`, the lightweight branch. Second, every component, including `JPanel`, is also a `Container` — something that can hold other components — which is how windows are built out of nested boxes.

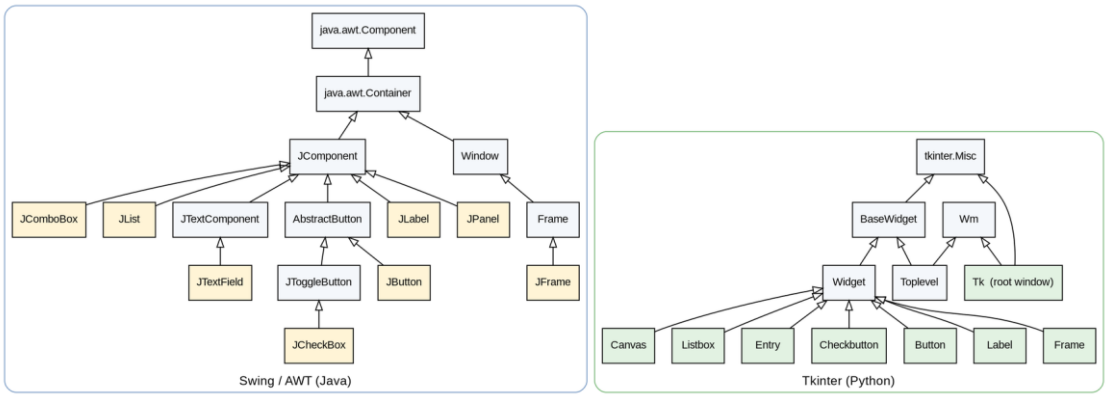


Figure 10.2 Part of the class hierarchies of Swing/AWT (left) and Tkinter (right). Shaded classes are the ones this chapter uses. Arrows point from a subclass to its superclass.

Other Java GUI libraries exist: the Eclipse IDE of Chapter 2 is built with SWT, which returns to native widgets, and JavaFX is a separately distributed toolkit (OpenJFX), released every six months like the JDK; JavaFX 27, the current version in September 2026, requires JDK 25 or later [2]. Swing is part of every JDK, including Java 25, the current long-term-support release, and Java 27, released on 15 September 2026 [3], which is why this book uses it. You may meet **applets**, small frames embedded in web pages, in older books; browsers dropped Java plug-ins long ago, the Applet API was deprecated for removal in Java 17 [4], and JDK 26 removed it altogether [5], so an applet no longer even compiles.

10.3.3 Tkinter: Python's standard toolkit

Python's standard library includes `tkinter`, the Python interface to **Tk**, a toolkit written in C in the early 1990s by John Ousterhout for his scripting language Tcl [6]. When a Python program creates a `tkinter.Button`, the Python object sends a command to an embedded Tcl interpreter, and Tk creates and draws the actual button. Like Swing, Tk draws most of its widgets itself and runs on Windows, macOS and Linux. The module `tkinter.ttk` adds *themed* widgets (such as the `Combobox` we will use for the list of services) that follow the look of the host system more closely [7].

On the right of **Figure 10.2** you can see that Tkinter's design is flatter than Swing's. Every widget class (`Label`, `Button`, `Entry`, `Frame`, `Canvas`, …) inherits from `Widget`, which provides the common methods — `pack`, `grid`, `bind`, `config`, `destroy` and many more — through the mixin class `Misc`. The root window, class `Tk`, is special: there is exactly one per program, it owns the Tcl interpreter, and every other widget is created *inside* some parent, which is always the first argument of its constructor.

> ⚠ **WARNING.** Tkinter is part of the standard library, but on some systems it is packaged separately. The installers from python.org for Windows and macOS include it. On Debian and Ubuntu Linux, install it with `sudo apt install python3-tk`; on Fedora, `sudo dnf install python3-tkinter`. If `python3 -c "import tkinter"` prints `ModuleNotFoundError: No module named 'tkinter'`, this is the cause, not your code.

Table 10.1 pairs the components of the two toolkits that this chapter uses. The names differ; the ideas are the same.

Table 10.1 Common components of Swing and their Tkinter counterparts.

What it is	Swing (Java)	Tkinter (Python)
Top-level window	`JFrame`	`Tk` (the root), `Toplevel` (others)
Plain container	`JPanel`	`Frame`, `LabelFrame`
Fixed text or image	`JLabel`	`Label`
Push button	`JButton`	`Button`
One-line text input	`JTextField`	`Entry`
Tick box	`JCheckBox`	`Checkbutton`
One of several (radio)	`JRadioButton` + `ButtonGroup`	`Radiobutton` sharing one variable
Drop-down choice	`JComboBox<E>`	`ttk.Combobox`
List of rows	`JList<E>` + `DefaultListModel<E>`	`Listbox`
Scrolling	`JScrollPane` around a component	`Scrollbar` linked to a widget
Free drawing	a `JPanel` subclass overriding `paintComponent`	`Canvas` with drawing items
Menus	`JMenuBar`, `JMenu`, `JMenuItem`	`Menu`
Tabs	`JTabbedPane`	`ttk.Notebook`

The catalogue of each toolkit is much longer. Beyond display elements (labels, images) and basic controls (fields, buttons, radio buttons, check boxes, combo boxes, menus, sliders), there are containers (panels, scroll panes, split panes, tabbed panes, tool bars), fancy displays (tables and trees) and fancy controls (file choosers and color choosers). The official Swing tutorial [1] and the `tkinter` documentation [7] list them all; once you understand the ideas of this chapter, each new component is a matter of reading its reference page.

10.4 Your First Window

We begin with the smallest useful window: a frame with a title and one label that says hello. **Listing 10.1** is the Swing version.

Listing 10.1 `FirstWindow.java`: a frame with one label, built in six steps, with a `--selftest` mode that runs without a screen.

```
import java.awt.BorderLayout;
import java.awt.Container;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;

public class FirstWindow {
    public static void main(String[] args) throws Exception {
        if (args.length > 0 && args[0].equals("--selftest")) {
            System.setProperty("java.awt.headless", "true");  // no screen needed
            SwingUtilities.invokeAndWait(FirstWindow::selfTest);
        } else {
            SwingUtilities.invokeLater(FirstWindow::createAndShow);
        }
    }

    static JLabel buildLabel() {
        return new JLabel("Hello, Profitina Laundry!");
    }

    static void createAndShow() {
        // 1. Optional: let Swing draw the window decorations
        JFrame.setDefaultLookAndFeelDecorated(true);
        // 2. Create the frame
        JFrame frame = new JFrame("First window");
        // 3. Optional: what happens when the frame closes?
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // 4. Create components and put them in the frame
        frame.getContentPane().add(buildLabel(), BorderLayout.CENTER);
        // 5. Size the frame to fit its contents
        frame.pack();
        // 6. Show it
        frame.setVisible(true);
    }

    static void selfTest() {
        Container content = new Container();  // stands in for the content pane
        content.setLayout(new BorderLayout());
        JLabel label = buildLabel();
        content.add(label, BorderLayout.CENTER);
        System.out.println("headless: " + java.awt.GraphicsEnvironment.isHeadless());
        System.out.println("components: " + content.getComponentCount());
        System.out.println("label text: " + label.getText());
        System.out.println("on the EDT: " + SwingUtilities.isEventDispatchThread());
        System.out.println("self-test passed");
    }
```

Listing 10.1 (Continued) *FirstWindow.java: a frame with one label, built in six steps, with a --selftest mode that runs without a screen.*

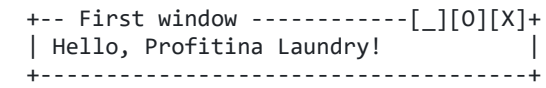
```
}
```

The method `createAndShow` contains the six steps that almost every Swing program performs, numbered in the comments:

1. `JFrame.setDefaultLookAndFeelDecorated(true)` is optional. It asks Swing to draw the frame's title bar and borders itself, in Swing's own look, instead of letting the operating system draw them. The windows in this chapter's figures that have a patterned blue title bar were made this way.
2. `new JFrame("First window")` creates the frame object, with the given title. Creating it does not show it.
3. `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` says what should happen when the user clicks the close button: end the program. The default is merely to *hide* the frame, which leaves the program running invisibly, with no window to close — the program appears to hang. Always write this line for the main window.
4. `frame.getContentPane().add(buildLabel(), BorderLayout.CENTER)` puts a component into the frame. A `JFrame` does not hold components directly; it holds several *panes*, and the one that holds your components is the **content pane**. Section 10.5 explains the panes and the `BorderLayout.CENTER` argument.
5. `frame.pack()` sizes the frame so that it is just big enough for its contents at their *preferred sizes*. The alternative, `setSize(width, height)`, fixes the size in pixels whatever the contents need; `pack()` is almost always better, because the preferred size of a label depends on its font, which differs from one computer to another.
6. `frame.setVisible(true)` shows the frame. Until this call, the window exists only as objects in memory.

Now look at `main`. It does not call `createAndShow()` directly. It passes the method to `SwingUtilities.invokeLater`, which means "run this, soon, on the thread that handles the GUI". That thread, the **event dispatch thread** (EDT), is the subject of Section 10.11; for now take it as a rule that every Swing program's `main` creates its window this way. After `invokeLater` returns, `main` itself ends — and yet the program keeps running, because the EDT keeps running as long as a window is open. That is the first visible sign of inversion of control: `main` is no longer where the program lives.

The window it shows is tiny:



A title bar with the title and the three usual buttons, and underneath, the label, left-aligned and vertically centered in a frame that `pack()` made just wide enough for it.

10.4.1 Running a window without a screen: the self-test

This book cannot show you a live window, and neither can an automated test running on a server with no monitor. So every GUI program in this chapter has a second mode. Run with the argument `--selftest`, it builds the window's components exactly as the real program does, drives them from code, prints what it finds, and never opens a window at all. For **Listing 10.1**, the command `java FirstWindow --selftest` prints:

```
headless: true
components: 1
label text: Hello, Profitina Laundry!
on the EDT: true
self-test passed
```

The first line of `main` in self-test mode, `System.setProperty("java.awt.headless", "true")`, switches AWT into **headless mode**: the mode for computers without a display. In headless mode

you may create and use any lightweight component — labels, panels, buttons, text fields, lists — but anything that needs a real native window, such as a `JFrame`, throws `HeadlessException`. That is why the self-test puts the label into a plain `Container` that stands in for the content pane, instead of into a frame. The property must be set before the first AWT class is used, which is why it is the very first statement.

Try running the program *normally* on a machine with no display (for example over SSH, or in a container) and you see what the self-test avoids:

```
Exception in thread "AWT-EventQueue-0" java.awt.HeadlessException:
No X11 DISPLAY variable was set,
or no headful library support was found,
but this program performed an operation which requires it.
```

Notice the thread name in that message, `AWT-EventQueue-0`: the exception happened on the event dispatch thread, where `invokeLater` sent our `createAndShow`. The self-test used `invokeAndWait` instead, which runs the code on the EDT *and waits for it to finish*, so that `main` does not end before the test has printed its results. The line `on the EDT: true` confirms where it ran.

Listing 10.2 is the same program in Tkinter.

Listing 10.2 `first_window.py`: the same window in Tkinter, with the same `--selftest` mode.

```
"""first_window.py: the smallest useful Tkinter window."""
import sys
import tkinter as tk


def build(root):
    root.title("First window")
    label = tk.Label(root, text="Hello, Profitina Laundry!", padx=20, pady=10)
    label.pack()
    return label


def self_test():
    root = tk.Tk()
    root.withdraw()  # build the window, but never show it
    label = build(root)
    root.update_idletasks()  # let the geometry manager do its work
    print("title:", root.title())
    print("children:", [w.winfo_class() for w in root.winfo_children()])
    print("label text:", label.cget("text"))
    print("mapped on screen:", bool(root.winfo_ismapped()))
    root.destroy()
    print("self-test passed")


def main():
    if "--selftest" in sys.argv:
        self_test()
        return
    root = tk.Tk()
    build(root)
    root.mainloop()  # hand control to the event loop until the window closes


if __name__ == "__main__":
    main()
```

The structure maps one to one. `tk.Tk()` creates the root window (Swing's frame), `root.title(...)` sets the title, `tk.Label(root, text=...)` creates a label *inside* the root — the

parent is always the first argument — and `label.pack()` hands the label to the **pack geometry manager**, which places it and sizes the window to fit, much like `pack()` in Swing. There is no step 3: closing Tkinter's root window ends `mainloop()`, and with it the program. And the last line of `main`, `root.mainloop()`, is where Python hands control to Tk's event loop, the equivalent of Swing's EDT. `mainloop()` returns only when the window is closed.

Running `python3 first_window.py --selftest` prints:

```
title: First window
children: ['Label']
label text: Hello, Profitina Laundry!
mapped on screen: False
self-test passed
```

The self-test calls `root.withdraw()` immediately after creating the root, which removes the window from the screen before it is ever drawn. `update_idletasks()` lets Tk finish its pending bookkeeping (including geometry calculations) without processing any user events. The line `mapped on screen: False` confirms that nothing was shown.

There is one difference between the toolkits that matters for testing. Swing has a genuine headless mode. Tk does not: even a withdrawn window needs a connection to a display, and on a machine without one, `tk.Tk()` fails with

```
_tkinter.TclError: no display name and no $DISPLAY environment variable
```

On a Linux server the standard remedy is **Xvfb**, the X virtual frame buffer, a display server that draws into memory instead of onto a monitor. The command `xvfb-run -a python3 first_window.py --selftest` starts one, runs the program against it, and shuts it down. Every Tkinter output in this chapter was produced that way; on your own computer, with a screen, you simply run the command without `xvfb-run`.

> ✓ **KEY INSIGHT.** A self-test mode costs a dozen lines and pays for itself many times over. It forces you to build the window in a method that can be called without showing anything, it lets you check the window's behavior from code, and it lets a test server (or a book) run your GUI program. Chapter 11 takes the idea further by separating the window completely from the logic it displays.

10.5 Components, Containers, and Panels

The things we can display in a GUI window are called **components** (Java) or **widgets** (Tk): labels, buttons, text fields, tables, and so on. Some components are **containers**: they can contain other components, which may themselves be containers, and so on to any depth. A window is therefore not a flat collection of controls but a *tree*: the frame at the root, containers as inner nodes, and simple controls as leaves.

10.5.1 The content pane

A `JFrame` is a container of a specialized sort. It does not hold your components directly; it has several *panes* as its components — a root pane, a layered pane used for pop-up menus and tool tips, an invisible glass pane that can intercept the mouse, and the one you care about, the **content pane**. The content pane is an ordinary container, so there are two ways to use it. You can add components to the one the frame already has:

```
Container pane = frame.getContentPane();
pane.add(new JLabel("Hello world!"), BorderLayout.PAGE_START);
pane.add(new JButton("Click me"), BorderLayout.PAGE_END);
```

or you can build your own container and install it as the new content pane:

```
JPanel pane = new JPanel(new BorderLayout());
pane.add(new JLabel("Hello world!"), BorderLayout.PAGE_START);
pane.add(new JButton("Click me"), BorderLayout.PAGE_END);
frame.setContentPane(pane);
```

The second form is the one this chapter uses for every real window, because it lets the content be a class of our own. As a shortcut, `frame.add(c)` is forwarded to the content pane, so `frame.add(panel)` and `frame.getContentPane().add(panel)` do the same thing.

10.5.2 Panels, and why every window here is a class

A **panel**, class `JPanel`, is the simplest lightweight container: a rectangle that can hold components and that can also be drawn on (Section 10.9). It is good design practice to define each significant part of a window as a *class that extends* `JPanel`, putting its components in fields and building it in the constructor:

```
public class MyPanel extends JPanel {
    int count = 0;                              // the state
    JButton button = new JButton("Click me");
    JLabel label = new JLabel("This is a label");

    public MyPanel() {
        add(button);                            // this.add(...): the panel adds to
itself
        add(label);
    }
}
```

and then, in `main`, to install an instance of it as a frame's content pane with `frame.setContentPane(new MyPanel())`. This is Chapter 7's inheritance at work: `MyPanel` *is a* `JPanel`, so it inherits everything a panel can do — be added to a frame, lay out children, paint itself — and adds only its own components and state. When we need custom drawing, extending `JPanel` is not merely good practice but necessary, because drawing is done by overriding one of its methods.

Tkinter programs follow the same pattern with `tk.Frame`: `class EchoPanel(tk.Frame)` calls `super().__init__(master)` and then creates its widgets with `self` as their parent. Every panel in this chapter, in both languages, is built this way.

10.5.3 The component tree of a real window

Figure 10.3 shows the tree of the laundry window that we build in Section 10.12. The frame holds the content pane; the content pane, a `LaundryOrderWindow`, holds three containers — the form at the top, the scrolling list in the middle, and a footer at the bottom — and each of those holds its own components. The form even contains a small panel of its own, just to keep the *Add order* and *Clear* buttons side by side. Reading a window as a tree is the key to laying it out: every container decides the positions of its own children and nothing else.

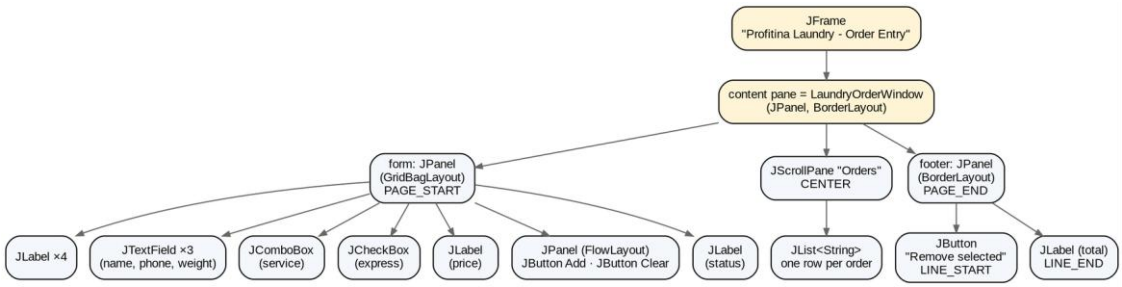


Figure 10.3 The component tree of the Profitina Laundry order-entry window. Each container arranges only its own children, with its own layout manager.

10.6 The Event Loop, Explained from Zero

We have said several times that a GUI program "hands control to a loop that waits for events". Before we write a single listener, let us take that loop apart, because once you have seen it, nothing else in this chapter is mysterious.

10.6.1 Where events come from

When you press the mouse button, the mouse hardware sends a signal to the computer. The operating system's input driver notices it, works out where the pointer is, and decides which window is under the pointer. It then puts a small record — "button 1 pressed at (x, y) in window W, at time t" — into a queue that belongs to the program owning window W. The same happens for key presses (the keyboard focus decides which window receives them), for mouse movements, for a window being uncovered and needing to be redrawn, and for many other happenings. Each such record is an **event**.

Inside the program, the toolkit reads these raw records and turns them into its own event objects — in Java, a `MouseEvent`, `KeyEvent`, `ActionEvent` and so on; in Tk, an event object with fields such as `x`, `y`, `keysym` and `widget`. It puts them into its own **event queue**, a first-in, first-out queue just like the `ArrayDeque` in the program of Section 10.6.3.

10.6.2 The loop itself

Somewhere inside every toolkit there is a loop that looks like **Pseudocode 10.1**.

Pseudocode 10.1 The event loop that every GUI toolkit runs: wait, take the next event, call the listeners registered for it, and repaint when idle.

```
EVENT-LOOP(queue)
1  while the program has not been told to quit
2      if queue is empty
3          wait until the operating system delivers an event
4      e = DEQUEUE(queue)
5      c = the component that e is aimed at
6      for each listener l registered on c for the kind of e
7          call l with e                 // your code runs here
8      if nothing is waiting and some region is marked "dirty"
9          repaint the dirty regions     // paintComponent runs here
```

Read it slowly. Line 3 is where a GUI program spends almost all of its life: doing nothing, not even using the processor, until the user acts. Lines 4–7 are **dispatch**: find out which component the event concerns, and call every listener that asked to hear about that kind of event on that component. Line 7 is the only place where your code ever runs after `main` has finished — and while it runs, the loop is stuck on line 7: no other event is dispatched and nothing is repainted.

Lines 8 and 9 are how the screen gets updated: code never draws on the screen directly; it marks a region as needing to be redrawn, and the loop redraws it when it has nothing more urgent to do.

Figure 10.4 draws the same loop as a picture, together with the route an event takes from the user's finger to your code.

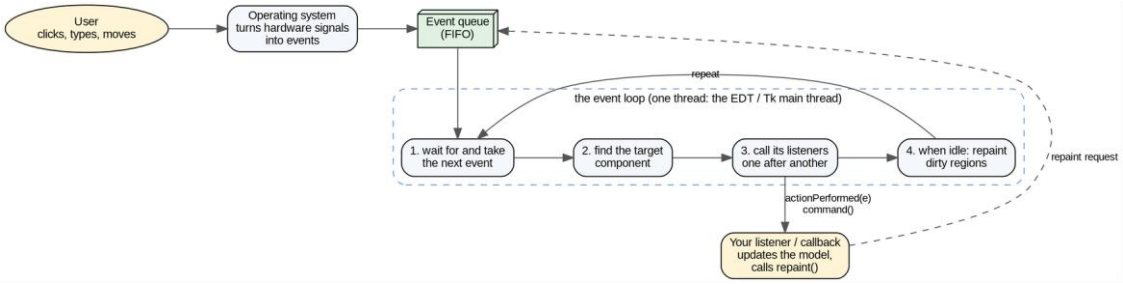


Figure 10.4 The life of an event. The operating system turns the user's actions into events and queues them; the event loop takes them one at a time, calls the listeners registered for them, and repaints when idle. A listener that changes what should be shown only *requests* a repaint.

10.6.3 An event loop you can read

The best way to believe the pseudocode is to run it. **Listing 10.3** and **Listing 10.4** build a complete miniature event system with no GUI library at all. There are pretend components (`Widget`), each with a list of listeners; a queue of events; and a loop that is **Pseudocode 10.1** almost line for line. The "user" types a weight of 4 kg, clicks a button, types 3 kg, clicks again, and closes the window; the button's listener adds Rp7,000 per kilogram to a total and asks for the total label to be redrawn.

Listing 10.3 `ToyEventLoop.java`: a hand-written event loop with pretend components, listeners and a queue.

```
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Queue;

public class ToyEventLoop {
    record Event(String target, String kind, String data) { }

    interface Listener {
        void handle(Event e);
    }

    static class Widget {  // a pretend component that keeps a list of listeners
        final String name;
        final List<Listener> listeners = new ArrayList<>();

        Widget(String name) {
            this.name = name;
        }

        void addListener(Listener l) {
            listeners.add(l);
        }

        void fire(Event e) {
            for (Listener l : listeners) {
                l.handle(e);
            }
        }
    }

    static final Queue<Event> queue = new ArrayDeque<>();
    static final Map<String, Widget> widgets = new LinkedHashMap<>();
    static int total = 0;
```

Listing 10.3 (Continued) *ToyEventLoop.java: a hand-written event loop with pretend components, listeners and a queue.*

```
    public static void main(String[] args) {
        Widget field = new Widget("weightField");
        Widget button = new Widget("addButton");
        Widget label = new Widget("totalLabel");
        for (Widget w : List.of(field, button, label)) {
            widgets.put(w.name, w);
        }
        final String[] typed = {""};  // what the field currently holds
        field.addListener(e -> typed[0] = e.data());
        button.addListener(e -> {
            total += Integer.parseInt(typed[0]) * 7000;
            queue.add(new Event("totalLabel", "repaint", "Rp" + total));
        });
        label.addListener(e -> System.out.println("    screen shows " + e.data()));

        // What the user will do, one action at a time:
        List<Event> userActions = List.of(
                new Event("weightField", "text", "4"),
                new Event("addButton", "click", ""),
                new Event("weightField", "text", "3"),
                new Event("addButton", "click", ""),
                new Event("window", "close", ""));
        int next = 0;

        // The event loop: wait for an event, dispatch it, repeat.
        while (true) {
            if (queue.isEmpty()) {  // nothing to do: wait for the user
                queue.add(userActions.get(next++));
            }
            Event e = queue.remove();
            System.out.println("loop: " + e.kind() + " -> " + e.target());
            if (e.kind().equals("close")) {
                break;
            }
            widgets.get(e.target()).fire(e);
        }
        System.out.println("loop ended; total = Rp" + total);
    }
}
```

Listing 10.4 `toy_event_loop.py`: the same event loop in Python, with callbacks as plain functions.

```
"""toy_event_loop.py: an event loop written by hand, with no GUI library."""
from collections import deque, namedtuple

Event = namedtuple("Event", "target kind data")


class Widget:  # a pretend component that keeps a list of callbacks
    def __init__(self, name):
        self.name = name
        self.listeners = []

    def add_listener(self, callback):
        self.listeners.append(callback)

    def fire(self, event):
        for callback in self.listeners:
            callback(event)


queue = deque()
widgets = {}
total = 0


def main():
    field = Widget("weightField")
    button = Widget("addButton")
    label = Widget("totalLabel")
    for w in (field, button, label):
        widgets[w.name] = w
    typed = [""]  # what the field currently holds

    def on_text(e):
        typed[0] = e.data

    def on_click(e):
        global total
        total += int(typed[0]) * 7000
        queue.append(Event("totalLabel", "repaint", f"Rp{total}"))

    field.add_listener(on_text)
    button.add_listener(on_click)
    label.add_listener(lambda e: print("    screen shows", e.data))
```

Listing 10.4 (Continued) *toy_event_loop.py: the same event loop in Python, with callbacks as plain functions.*

```
    # What the user will do, one action at a time:
    user_actions = deque([Event("weightField", "text", "4"),
                          Event("addButton", "click", ""),
                          Event("weightField", "text", "3"),
                          Event("addButton", "click", ""),
                          Event("window", "close", "")])

    # The event loop: wait for an event, dispatch it, repeat.
    while True:
        if not queue:  # nothing to do: wait for the user
            queue.append(user_actions.popleft())
        e = queue.popleft()
        print(f"loop: {e.kind} -> {e.target}")
        if e.kind == "close":
            break
        widgets[e.target].fire(e)
    print(f"loop ended; total = Rp{total}")


if __name__ == "__main__":
    main()
```

Both programs print exactly the same lines:

```
loop: text -> weightField
loop: click -> addButton
loop: repaint -> totalLabel
    screen shows Rp28000
loop: text -> weightField
loop: click -> addButton
loop: repaint -> totalLabel
    screen shows Rp49000
loop: close -> window
loop ended; total = Rp49000
```

Match the output to **Pseudocode 10.1**. The queue starts empty, so the loop "waits" — here, it takes the user's next scripted action, which stands in for line 3. The first event is the text "4" arriving at the weight field; the field's only listener stores it. Next comes the click; the button's listener multiplies $4 \times 7000 = 28{,}000$, adds it to the total and — this is the important part — does *not* print anything itself. It *enqueues a repaint event* for the label. The loop's next pass finds that repaint event already in the queue, so it dispatches it before asking the user for anything else, and the label's listener "shows" Rp28000 on the "screen". The same happens for the 3 kg bag ($3 \times 7000 = 21{,}000$, total 49,000). Finally the close event breaks the loop, and only then does `main` continue past the loop and print the last line.

Three details of the code deserve attention now, because they reappear in every real toolkit.

- **Registration.** `button.addListener(e -> { … })` (Java) and `button.add_listener(on_click)` (Python) do not run anything. They only store the listener in the widget's list, to be called later. Beginners often expect the code inside a listener to run at the point where it is registered; it never does.
- **Dispatch by target.** The loop does not know what any listener does. It looks up the widget by the event's target name and calls `fire`, which calls each registered listener in the order of registration. A widget may have any number of listeners, including none.
- **Listeners share state through objects.** The field's listener and the button's listener communicate through `typed`, a one-element array in Java and a one-element list in Python. Why not a plain `String typed`? In Java, a lambda may only use local variables that never change after they are assigned (Section 10.10 explains why), so we let the lambda change the *contents* of an

array instead of the variable itself. In Python, a nested function may read a variable of the enclosing function, but assigning to it would create a new local variable unless it is declared `nonlocal`; again the one-element list sidesteps the issue. In a real program, as in the rest of this chapter, the shared state simply lives in fields of the panel object.

Real toolkits add many refinements — events for every kind of hardware, coalescing of repeated repaints into one, timers, keyboard focus, priorities — but the skeleton is **Listing 10.3**. In Swing, the loop runs on the event dispatch thread, which AWT starts automatically the first time a component is shown or `invokeLater` is called. In Tkinter, the loop is `mainloop()`, running on the thread that called it — normally the main thread.

10.7 Listening to Events

We can now read and write real event handling. In Java, for a program to listen to any type of event it must do three things:

1. **declare** that it listens to that type of event, by implementing the corresponding *listener interface* (for button clicks, `ActionListener`);
2. **register** itself with every component that can trigger the event, by calling that component's `add…Listener` method (for a button, `addActionListener`);
3. **define** the method or methods that the interface requires (for `ActionListener`, the single method `actionPerformed(ActionEvent e)`).

This is Chapter 6's interfaces put to work. The button does not know or care what class your listener is; it knows only that the listener promised, by implementing `ActionListener`, to have a method `actionPerformed`. When the button is clicked, the event loop asks the button to notify its listeners, and the button calls that method on each of them. **Figure 10.5** shows the arrangement: a button holds a list of listeners, `addActionListener` appends to the list, and a click calls every listener in turn. The same design, a subject that notifies a list of observers without knowing what they do, is the **Observer** pattern [8], and it is the backbone of every GUI toolkit.

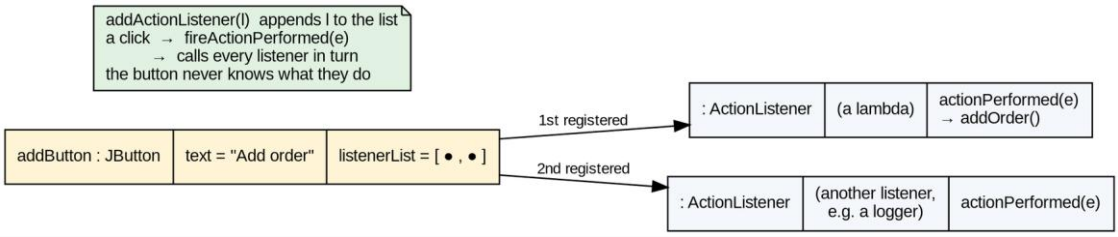


Figure 10.5 A button and its listeners. The button keeps a list; registration appends to it; a click calls each listener's `actionPerformed` in turn. The button never knows what its listeners do.

10.7.1 Reading a text field

Our first real listener reads what the user typed. Users enter text in text-field components; a `JTextField` fires an *action event* when the user presses Enter (Return) inside it. **Listing 10.5** is a panel with a text field and a label: when the user presses Enter, the label is replaced by whatever was typed. The four numbered comments mark the four changes compared with a panel that merely displays components.

Listing 10.5 `EchoPanel.java`: a panel that listens for Enter in its text field and copies the text into a label.

```
import java.awt.event.ActionEvent;       // (1) the event-handling library
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class EchoPanel extends JPanel
        implements ActionListener {         // (2) declare: this panel listens
    private final JTextField t = new JTextField(15);
    private final JLabel l = new JLabel("Write something on text-field & enter");

    public EchoPanel() {
        add(t);
        t.addActionListener(this);          // (3) register with the text field
        add(l);
    }

    @Override
    public void actionPerformed(ActionEvent e) {  // (4) define the method
        l.setText(t.getText());
    }

    public static void main(String[] args) throws Exception {
        if (args.length > 0 && args[0].equals("--selftest")) {
            System.setProperty("java.awt.headless", "true");
            SwingUtilities.invokeAndWait(EchoPanel::selfTest);
        } else {
            SwingUtilities.invokeLater(EchoPanel::createAndShow);
        }
    }

    static void createAndShow() {
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame frame = new JFrame("JFrame demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setContentPane(new EchoPanel());
        frame.pack();
        frame.setVisible(true);
    }
```

Listing 10.5 (Continued) *EchoPanel.java: a panel that listens for Enter in its text field and copies the text into a label.*

```
    static void selfTest() {
        EchoPanel panel = new EchoPanel();
        System.out.println("before: " + panel.l.getText());
        panel.t.setText("Hi, I'm Yarik. Nice to meet you!");
        System.out.println("typed:  " + panel.t.getText());
        panel.t.postActionEvent();          // exactly what pressing Enter does
        System.out.println("after:  " + panel.l.getText());
        System.out.println("listeners on the field: "
                + panel.t.getActionListeners().length);
    }
}
```

- (1) `import java.awt.event.ActionEvent` and `ActionListener`: the event-handling classes live in `java.awt.event`, not in `javax.swing`. Many programs write `import java.awt.event.*;` instead.

(2) `implements ActionListener` declares that an `EchoPanel` *is an* `ActionListener`: it can handle action events.

- (3) `t.addActionListener(this)` registers *this* panel with the text field `t`: "when an action happens in `t`, call my `actionPerformed`". The keyword `this` means the object being constructed — the panel itself.
- (4) `actionPerformed(ActionEvent e)` is the method the interface requires. It calls `t.getText()` to read the text currently in the field and `l.setText(...)` to change the label.

The `JTextField(15)` constructor makes a field wide enough for about 15 characters; the width affects only the preferred size, not how much may be typed. The panel uses `JPanel`'s default layout, `FlowLayout`, which puts the field and the label side by side.

Figure 10.6 shows the window after the user typed a greeting and pressed Enter: the label, which used to say *Write something on text-field & enter*, now repeats the greeting.

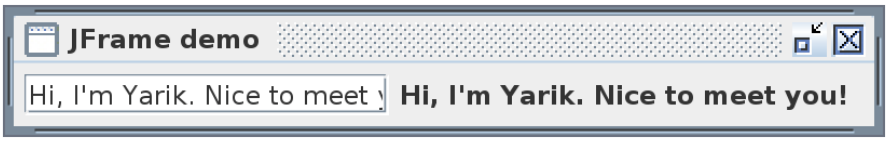


Figure 10.6 The echo window after the user typed a greeting into the text field and pressed Enter: the listener has copied the text into the label.

The self-test proves the same thing without a screen:

```
before: Write something on text-field & enter
typed:  Hi, I'm Yarik. Nice to meet you!
after:  Hi, I'm Yarik. Nice to meet you!
listeners on the field: 1
```

The key line is `panel.t.postActionEvent()`. `JTextField.postActionEvent()` is exactly what the field does internally when Enter is pressed: it builds an `ActionEvent` and delivers it to every registered listener. The last line confirms that one listener is registered — our panel. Notice that the test calls no method of `EchoPanel` directly; it drives the components, and the components call the panel, just as they would for a real user.

Listing 10.6 is the Tkinter version.

Listing 10.6 `echo_panel.py`: the same echo panel in Tkinter, using `bind` to attach a callback to the Enter key.

```
"""echo_panel.py: copy the text of an entry into a label when Enter is pressed."""
import sys
import tkinter as tk


class EchoPanel(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.t = tk.Entry(self, width=15)
        self.t.pack(side=tk.LEFT, padx=4, pady=4)
        self.t.bind("<Return>", self.on_enter)  # register the callback
        self.l = tk.Label(self, text="Write something on text-field & enter")
        self.l.pack(side=tk.LEFT, padx=4)

    def on_enter(self, event):  # the callback: Tk passes an event object
        self.l.config(text=self.t.get())


def self_test():
    root = tk.Tk()
    root.withdraw()
    panel = EchoPanel(root)
    panel.pack()
    print("before:", panel.l.cget("text"))
    panel.t.insert(0, "Hi, I'm Yarik. Nice to meet you!")
    print("typed: ", panel.t.get())
    panel.on_enter(None)  # what Tk calls when Enter is pressed on screen
    print("after: ", panel.l.cget("text"))
    print("bindings on the entry:", panel.t.bind())
    root.destroy()


def main():
    if "--selftest" in sys.argv:
        self_test()
        return
    root = tk.Tk()
    root.title("JFrame demo")
    EchoPanel(root).pack()
    root.mainloop()


if __name__ == "__main__":
    main()
```

```
before: Write something on text-field & enter
typed:  Hi, I'm Yarik. Nice to meet you!
after:  Hi, I'm Yarik. Nice to meet you!
bindings on the entry: ('<Key-Return>',)
```

Tkinter does not use listener interfaces. Instead, `widget.bind(sequence, callback)` attaches any callable to an **event sequence**, written as a string in angle brackets: `"<Return>"` is the Enter key, `"<Button-1>"` a press of the left mouse button, `"<B1-Motion>"` moving the mouse with the left button held down, `"<KeyPress-a>"` the A key. When the event occurs over (or, for keys, with the focus in) that widget, Tk calls the callback with one argument, an **event object** whose attributes describe what happened: `event.widget`, `event.x` and `event.y`, `event.keysym`, and so on. `self.on_enter` is a *bound method* — the method together with the object it belongs to — so when Tk calls it with the event, Python supplies `self` automatically. The last line of the test shows how Tk reports the binding: `'<Key-Return>'` is the full name of `"<Return>"`.

The Tkinter self-test differs in one telling way: it calls `panel.on_enter(None)` itself. Tk delivers keyboard events only to a window that is visible on the screen and has the keyboard focus; a

withdrawn window receives none, and `event_generate("<Return>")` on it silently does nothing (try it). So the test does what Tk would do for a real key press — call the bound callback — and passes `None` for the event object, which `on_enter` does not use. For buttons and check buttons Tk offers `invoke()`, which works without a screen, and we will use it from now on.

10.7.2 The event object, and which component caused the event

Every listener method receives an event object, and the information in it depends on the type of event. You can always ask a *component* for its state (`t.getText()`, `cb.isSelected()`); the *event* tells you what just happened:

- for action events and item events, `e.getSource()` returns the component that fired the event — which button, which text field, which check box;
- for mouse events, `e.getPoint()` returns the position of the pointer as a `Point` whose fields `.x` and `.y` are the coordinates, and `e.getX()`, `e.getY()` return them directly;
- for key events, `e.getKeyCode()` and `e.getKeyChar()` say which key.

`getSource()` matters whenever one listener is registered with several components, because it is the only way the listener can tell them apart. Section 10.9 uses it to distinguish a *Darken* button from a *Lighten* button.

10.7.3 Item events: check boxes

A check box can be ticked and unticked; each change of state is an **item event**. The three steps are the same, with different names: declare `implements ItemListener`, register with `cb.addItemListener(this)`, and define `itemStateChanged(ItemEvent e)`. **Listing 10.7** has two check boxes and one listener for both.

Listing 10.7 `FruitChoice.java`: two check boxes and one item listener that reports which box changed and how.

```
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class FruitChoice extends JPanel implements ItemListener {
    private final JLabel label = new JLabel("Which one do you like?");
    private final JCheckBox cb1 = new JCheckBox("Mango");
    private final JCheckBox cb2 = new JCheckBox("Apple");
    private final JLabel explanation = new JLabel("You haven't checked anything");

    public FruitChoice() {
        add(label);
        add(cb1);
        add(cb2);
        add(explanation);
        cb1.addItemListener(this);
        cb2.addItemListener(this);
    }

    @Override
    public void itemStateChanged(ItemEvent e) {
        JCheckBox cb3 = (JCheckBox) e.getItemSelectable();  // which box changed?
        if (cb3.isSelected()) {
            explanation.setText(cb3.getText() + " is checked");
        } else {
            explanation.setText(cb3.getText() + " is unchecked");
        }
    }

    public static void main(String[] args) throws Exception {
        if (args.length > 0 && args[0].equals("--selftest")) {
            System.setProperty("java.awt.headless", "true");
            SwingUtilities.invokeAndWait(FruitChoice::selfTest);
        } else {
            SwingUtilities.invokeLater(() -> {
                JFrame.setDefaultLookAndFeelDecorated(true);
                JFrame frame = new JFrame("Item event");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.add(new FruitChoice());
                frame.pack();
                frame.setVisible(true);
            });
```

Listing 10.7 (Continued) *FruitChoice.java: two check boxes and one item listener that reports which box changed and how.*

```
        }
    }

    static void selfTest() {
        FruitChoice panel = new FruitChoice();
        System.out.println("start:        " + panel.explanation.getText());
        panel.cb1.doClick();  // the same path as a real mouse click
        System.out.println("click Mango:  " + panel.explanation.getText());
        panel.cb1.doClick();
        System.out.println("click Mango:  " + panel.explanation.getText());
        panel.cb2.doClick();
        System.out.println("click Apple:  " + panel.explanation.getText());
        panel.cb2.doClick();
        System.out.println("click Apple:  " + panel.explanation.getText());
    }
}
```

`e.getItemSelectable()` returns the component whose state changed (for a check box, the same object as `getSource()`, but typed as an `ItemSelectable`); the cast `(JCheckBox)` gives us access to `isSelected()` and `getText()`. **Figure 10.7** shows the window after the user ticked *Mango*.



Figure 10.7 The check-box window after *Mango* was ticked: the item listener rewrote the explanation label.

The self-test clicks each box twice with `doClick()`, which performs a click programmatically through exactly the same path as a mouse click, including firing the listeners:

```
start:        You haven't checked anything
click Mango:  Mango is checked
click Mango:  Mango is unchecked
click Apple:  Apple is checked
click Apple:  Apple is unchecked
```

Listing 10.8 is the Tkinter version.

Listing 10.8 `fruit_choice.py`: the same choice in Tkinter, with a `BooleanVar` per check button and a `command` callback.

```
"""fruit_choice.py: two check buttons that report which one changed."""
import sys
import tkinter as tk


class FruitChoice(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        tk.Label(self, text="Which one do you like?").pack(side=tk.LEFT)
        self.mango = tk.BooleanVar(value=False)
        self.apple = tk.BooleanVar(value=False)
        self.cb1 = tk.Checkbutton(self, text="Mango", variable=self.mango,
                                  command=lambda: self.changed("Mango", self.mango))
        self.cb2 = tk.Checkbutton(self, text="Apple", variable=self.apple,
                                  command=lambda: self.changed("Apple", self.apple))
        self.cb1.pack(side=tk.LEFT)
        self.cb2.pack(side=tk.LEFT)
        self.explanation = tk.Label(self, text="You haven't checked anything")
        self.explanation.pack(side=tk.LEFT)

    def changed(self, name, var):  # the lambdas above tell us which box changed
        if var.get():
            self.explanation.config(text=name + " is checked")
        else:
            self.explanation.config(text=name + " is unchecked")


def self_test():
    root = tk.Tk()
    root.withdraw()
    panel = FruitChoice(root)
    panel.pack()
    print("start:       ", panel.explanation.cget("text"))
    for box, name in [(panel.cb1, "Mango"), (panel.cb1, "Mango"),
                      (panel.cb2, "Apple"), (panel.cb2, "Apple")]:
        box.invoke()  # the same path as a real mouse click
        print(f"click {name}: ", panel.explanation.cget("text"))
    root.destroy()
```

Listing 10.8 (Continued) *fruit_choice.py: the same choice in Tkinter, with a BooleanVar per check button and a command callback.*

```
def main():
    if "--selftest" in sys.argv:
        self_test()
        return
    root = tk.Tk()
    root.title("Item event")
    FruitChoice(root).pack(padx=6, pady=6)
    root.mainloop()


if __name__ == "__main__":
    main()
```

```
start:       You haven't checked anything
click Mango:  Mango is checked
click Mango:  Mango is unchecked
click Apple:  Apple is checked
click Apple:  Apple is unchecked
```

The Tkinter version shows two ideas we will use again. First, a Tkinter `Checkbutton` stores its state in a **control variable**, here a `BooleanVar`: ticking the box sets the variable to `True`, and

`var.get()` reads it. Control variables (`StringVar`, `IntVar`, `DoubleVar`, `BooleanVar`) can be shared by several widgets and can be *watched* for changes, which the laundry window will use for its live price. Second, instead of one listener asking "which box was it?", each check button gets its own `command`, a small `lambda` that passes the box's name and variable to a shared method. The lambda *is* the answer to "which box?": each one remembers its own arguments. Section 10.10 explains exactly how.

10.7.4 The four kinds of events in this chapter

Table 10.2 summarizes, for the four major event types, what a Java program must declare, how it registers, and which methods it must define — and what the equivalent is in Tkinter. Mouse and mouse-motion events need no component to register with when the panel listens to itself: the panel simply calls `addMouseListener(this)` on itself, because the panel is the component the mouse is over.

Table 10.2 The four major event types: how to listen to them in Swing and in Tkinter.

Event	Java: declare	Java: register	Java: methods to define	Tkinter
Action (button clicked, Enter in a field)	`implements ActionListener`	`c.addActionListener(l)`	`actionPerformed(ActionEvent)`	`command=f`, or `bind("<Return>", f)`
Item (checkbox ticked or unticked)	`implements ItemListener`	`c.addItemListener(l)`	`itemStateChanged(ItemEvent)`	`command=f` on a `Checkbutton` with a variable
Mouse (press, release, click, enter, exit)	`implements MouseListener`	`addMouseListener(l)`	`mousePressed`, `mouseReleased`, `mouseClicked`, `mouseEntered`, `mouseExited` (all `MouseEvent`)	`bind("<ButtonPress-1>", f)`, `"<ButtonRelease-1>"`, `"<Enter>"`, `"<Leave>"`
Mouse motion (move, drag)	`implements MouseMotionListener`	`addMouseMotionListener(l)`	`mouseMoved`, `mouseDragged` (`MouseEvent`)	`bind("<Motion>", f)`, `bind("<B1-Motion>", f)`

A panel can listen to several types at once: it lists several interfaces after `implements`, registers for each, and defines all the required methods. `ShadeBox` in Section 10.9 does exactly that for action and mouse events.

10.8 Layout Managers versus pack and grid

Where should the *Add order* button go? One answer is "at pixel (192, 82), 104 pixels wide". That answer is wrong on the next computer, where the font is larger and the text no longer fits, and wrong again when the user makes the window wider, or when the program is translated into a language whose words are longer. Laying out GUI panels is a tricky business: the layout must allow for resizing of windows, and it must be flexible about the sizes of labels and buttons, which depend on fonts, languages and screen resolution.

Both toolkits therefore let you describe a layout by *rules* instead of coordinates. In Java, every container has a **layout manager**, an object that implements the interface `LayoutManager` and decides, whenever the container needs to be laid out, where each child goes and how big it is. It takes into account each child's *preferred*, *minimum* and *maximum* sizes (which the component computes from its font and contents) and the size of the container. In Tkinter, the same job is done by a **geometry manager**, of which there are three: `pack`, `grid` and `place`. Java's library defines a number of layout managers; you can also write your own or use third-party ones. We look at the four that cover almost every need — the official tutorial's visual guide shows them all [9] — and at `pack` and `grid`.

10.8.1 FlowLayout: like words in a paragraph

`FlowLayout` is the default layout manager of a `JPanel`. It places components left to right in a row, like words in a word processor, and when a row is full it starts a new one below. Components are kept at their preferred sizes as far as possible — no stretching or shrinking — and each row is centered horizontally by default. **Figure 10.8** shows five buttons in one row.



Figure 10.8 Five buttons in a `FlowLayout`: each keeps its preferred size, and the row flows from left to right.

10.8.2 BorderLayout: five regions

`BorderLayout` is the default for a frame's content pane. It divides the container into exactly five regions — `PAGE_START` (top), `PAGE_END` (bottom), `LINE_START` (left, in left-to-right languages), `LINE_END` (right) and `CENTER` — and each region holds at most one component. You choose the region when you add: `container.add(component, BorderLayout.CENTER)`. For other containers, set the layout first with `container.setLayout(new BorderLayout())` or pass it to the constructor. The five components stretch to fill all the available space: the top and bottom ones take the full width at their preferred height, the left and right ones take the remaining height at their preferred width, and the center gets everything left over. **Figure 10.9** shows the five regions. Of course, any of the five components can be a panel that groups smaller components together — which is exactly how the laundry window is built.

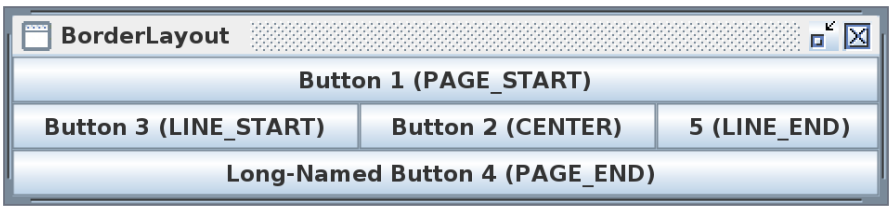


Figure 10.9 The five regions of a `BorderLayout`. The center takes all the space the four edges leave over.

10.8.3 BoxLayout: a single row or column

A **box** is an array of components laid out horizontally or vertically. `BoxLayout` arranges a container's children in one column (`BoxLayout.Y_AXIS`) or one row (`X_AXIS`): `panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS))`, or, through the `Box` container, `Box b = Box.createVerticalBox()`. Components can be aligned left, right or center, and invisible fillers space them out: `Box.createVerticalStrut(20)` is a fixed 20-pixel gap, `Box.createGlue()` a

spring that absorbs extra space. **Figure 10.10** shows two vertical boxes, each in a panel with a titled border, side by side in a horizontal box.

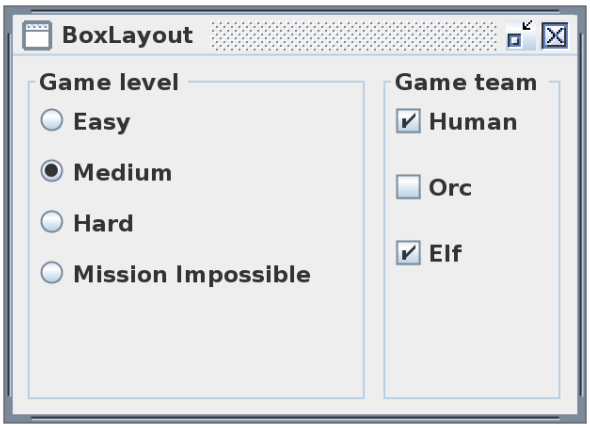


Figure 10.10 Two vertical boxes inside a horizontal box. The radio buttons on the left belong to one `ButtonGroup`, so only one can be chosen; the check boxes on the right are independent.

10.8.4 GridBagLayout: the most flexible

`GridBagLayout` is the most flexible layout manager in Swing, and the most verbose. Imagine the panel divided into a grid of invisible cells, whose rows and columns may have different sizes. Each component occupies a rectangular block of cells, and within its block it may stretch or not, be aligned to a side or a corner, and be padded. All these options are given in a separate object of class `GridBagConstraints`, which you fill in before each `add`:

```
panel.setLayout(new GridBagLayout());
GridBagConstraints c = new GridBagConstraints();
c.gridx = 0; c.gridy = 0;           // position: column 0, row 0
c.gridwidth = 1; c.gridheight = 3;  // size: one column wide, three rows high
c.weightx = 100; c.weighty = 100;   // stretch: share of any extra space
JList style = new JList();
panel.add(style, c);
```

The weights decide who receives extra space: the extra width of a row is shared among its components in proportion to `weightx`, and a weight of 0 means “never grow”. Typically the side columns keep their size (weight 0.0) and one column takes all the rest (weight 1.0). `fill` says whether a component grows to fill its cells (`NONE`, `HORIZONTAL`, `VERTICAL`, `BOTH`), `anchor` where it sits if it does not, `insets` how much margin surrounds it, and `gridwidth = GridBagConstraints.REMAINDER` makes a component the last in its row. **Figure 10.11** shows components spanning different numbers of cells.

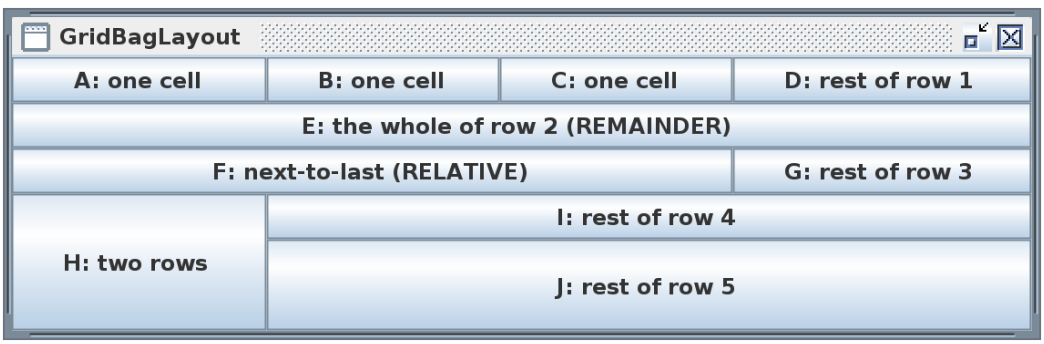


Figure 10.11 A `GridBagLayout` in which buttons span one cell, the rest of a row, or two rows, as set by `gridwidth`, `gridheight` and the weights.

Because the constraints object is reused from one `add` to the next, its fields keep their values: forgetting to reset `gridwidth` or a weight is the classic `GridBagLayout` bug. It is therefore good practice not to use `GridBagLayout` directly all over a program but to hide it in a small helper — a method or a `JPanel` subclass with a method such as `add(Component c, int x, int y, int w, int h)` that fills in the constraints for you. The laundry form of Section 10.12 does the same with a loop over its rows.

10.8.5 Tkinter: pack, grid and place

Tkinter's geometry managers are not objects owned by a container; they are methods called on each *child*, which also give it the rules for its placement.

- `widget.pack(side=..., fill=..., expand=..., padx=..., pady=...)` stacks widgets against the sides of their parent, in the order in which `pack` is called. `side=tk.LEFT` puts each widget to the right of the previous ones, like a row; `side=tk.TOP` (the default) puts each below the previous ones, like a column. `fill=tk.X` stretches a widget horizontally across the space it was given, and `expand=True` gives it a share of any extra space.
- `widget.grid(row=..., column=..., sticky=..., columnspan=..., padx=...)` puts widgets into the cells of a table, like `GridBagLayout` without the separate constraints object. `sticky="ew"` stretches a widget to the east and west edges of its cell (`"w"` alone aligns it to the left), and `parent.columnconfigure(1, weight=1)` gives column 1 all the extra width, just like `weightx`.
- `widget.place(x=..., y=...)` uses absolute coordinates and is rarely a good idea, for the reasons given at the start of this section.

Never mix `pack` and `grid` for children of the *same* parent. The two managers would fight over the parent's size; older versions of Tk simply froze while they negotiated, and Tk 8.6 refuses outright:

```
_tkinter.TclError: cannot use geometry manager grid inside . which already has slaves
managed by pack
```

(The `.` is Tk's name for the root window.) Using `pack` in one frame and `grid` in another is fine; the laundry window does exactly that.

10.8.6 Watching the managers work

Listing 10.9 builds the four kinds of panel. Run normally, it shows them on four tabs of a `JTabbedPane`. Run with `--selftest`, it gives each panel a size, asks the layout manager to lay it out *now* with `doLayout()`, and prints where every child ended up — so we can see the rules at work without a screen.

Listing 10.9 `LayoutTour.java`: the four main layout managers; the self-test prints the bounds each manager assigns.

```
import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.FlowLayout;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.Rectangle;
import javax.swing.AbstractButton;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class LayoutTour {
    static final String[] NAMES = {"Button 1", "Button 2", "Button 3",
                                   "Long-Named Button 4", "5"};

    static JPanel flow() {
        JPanel p = new JPanel(new FlowLayout());  // the default for JPanel
        for (String n : NAMES) {
            p.add(new JButton(n));
        }
        return p;
    }

    static JPanel border() {
        JPanel p = new JPanel(new BorderLayout());  // the default for a content pane
        p.add(new JButton("Button 1 (PAGE_START)"), BorderLayout.PAGE_START);
        p.add(new JButton("Button 2 (CENTER)"), BorderLayout.CENTER);
        p.add(new JButton("Button 3 (LINE_START)"), BorderLayout.LINE_START);
        p.add(new JButton("Long-Named Button 4 (PAGE_END)"), BorderLayout.PAGE_END);
        p.add(new JButton("5 (LINE_END)"), BorderLayout.LINE_END);
        return p;
    }
```

Listing 10.9 (Continued) *LayoutTour.java: the four main layout managers; the self-test prints the bounds each manager assigns.*

```
    static JPanel box() {
        JPanel p = new JPanel();
        p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));  // one column
        for (String n : NAMES) {
            JButton b = new JButton(n);
            b.setAlignmentX(Component.CENTER_ALIGNMENT);
            p.add(b);
        }
        return p;
    }

    static JPanel gridBag() {
        JPanel p = new JPanel(new GridBagLayout());
        GridBagConstraints c = new GridBagConstraints();
        c.insets = new Insets(2, 4, 2, 4);
        String[] labels = {"Customer", "Phone", "Weight (kg)"};
        for (int row = 0; row < labels.length; row++) {
            c.gridx = 0; c.gridy = row;              // position: column 0
            c.weightx = 0; c.fill = GridBagConstraints.NONE;
            c.anchor = GridBagConstraints.LINE_END;
            p.add(new JLabel(labels[row]), c);
            c.gridx = 1;                             // column 1 stretches
            c.weightx = 1; c.fill = GridBagConstraints.HORIZONTAL;
            p.add(new JTextField(10), c);
        }
        c.gridx = 1; c.gridy = labels.length;
        c.weightx = 0; c.fill = GridBagConstraints.NONE;
        c.anchor = GridBagConstraints.LINE_END;
        p.add(new JButton("Add order"), c);
        return p;
    }

    static String name(Component c) {
        if (c instanceof AbstractButton b) {
            return b.getText();
        }
        if (c instanceof JLabel l) {
            return l.getText();
        }
        return "[text field]";
    }
```

Listing 10.9 (Continued) *LayoutTour.java: the four main layout managers; the self-test prints the bounds each manager assigns.*

```
    static void report(String title, JPanel p, int w, int h) {
        p.setSize(w, h);
        p.doLayout();  // ask the layout manager to place the children now
        System.out.println(title + " in " + w + " x " + h + ":");
        for (Component c : p.getComponents()) {
            Rectangle r = c.getBounds();
            System.out.printf("  %-31s x=%3d y=%3d w=%3d h=%3d%n",
                    name(c), r.x, r.y, r.width, r.height);
        }
    }

    public static void main(String[] args) throws Exception {
        if (args.length > 0 && args[0].equals("--selftest")) {
            System.setProperty("java.awt.headless", "true");
            SwingUtilities.invokeAndWait(() -> {
                report("FlowLayout", flow(), 560, 40);
                report("FlowLayout", flow(), 230, 100);
                report("BorderLayout", border(), 460, 150);
                report("BoxLayout", box(), 230, 150);
                report("GridBagLayout", gridBag(), 300, 120);
            });
        } else {
            SwingUtilities.invokeLater(() -> {
                JFrame.setDefaultLookAndFeelDecorated(true);
                JFrame frame = new JFrame("Layout tour");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                JTabbedPane tabs = new JTabbedPane();
                tabs.addTab("Flow", flow());
                tabs.addTab("Border", border());
                tabs.addTab("Box", box());
                tabs.addTab("GridBag", gridBag());
                frame.add(tabs);
                frame.pack();
                frame.setVisible(true);
            });
        }
    }
}
```

```
FlowLayout in 560 x 40:
  Button 1                        x= 14 y=  5 w= 94 h= 25
  Button 2                        x=113 y=  5 w= 94 h= 25
  Button 3                        x=212 y=  5 w= 94 h= 25
  Long-Named Button 4             x=311 y=  5 w=187 h= 25
  5                               x=503 y=  5 w= 42 h= 25
FlowLayout in 230 x 100:
  Button 1                        x= 18 y=  5 w= 94 h= 25
  Button 2                        x=117 y=  5 w= 94 h= 25
  Button 3                        x= 68 y= 35 w= 94 h= 25
  Long-Named Button 4             x= 21 y= 65 w=187 h= 25
  5                               x= 94 y= 95 w= 42 h= 25
BorderLayout in 460 x 150:
  Button 1 (PAGE_START)           x=  0 y=  0 w=460 h= 25
  Button 2 (CENTER)               x=187 y= 25 w=153 h=100
  Button 3 (LINE_START)           x=  0 y= 25 w=187 h=100
  Long-Named Button 4 (PAGE_END)  x=  0 y=125 w=460 h= 25
  5 (LINE_END)                    x=340 y= 25 w=120 h=100
BoxLayout in 230 x 150:
  Button 1                        x= 67 y=  0 w= 94 h= 25
  Button 2                        x= 67 y= 25 w= 94 h= 25
  Button 3                        x= 67 y= 50 w= 94 h= 25
  Long-Named Button 4             x= 21 y= 75 w=187 h= 25
  5                               x= 93 y=100 w= 42 h= 25
GridBagLayout in 300 x 120:
  Customer                        x= 18 y= 15 w= 68 h= 15
  [text field]                    x= 94 y= 13 w=202 h= 19
  Phone                           x= 41 y= 38 w= 45 h= 15
  [text field]                    x= 94 y= 36 w=202 h= 19
  Weight (kg)                     x=  4 y= 61 w= 82 h= 15
  [text field]                    x= 94 y= 59 w=202 h= 19
  Add order                       x=192 y= 82 w=104 h= 25
```

Pixel values depend slightly on the fonts installed; the pattern does not. Read the output with the rules in mind.

- **FlowLayout, 560 pixels wide.** All five buttons fit in one row at their preferred sizes, 25 pixels high, with 5-pixel gaps. The row is 531 pixels wide in total ($503 + 42 - 14$), so the manager centered it by starting at $x = 14$.
- **FlowLayout, 230 pixels wide.** The same buttons no longer fit, so the manager wrapped them into four rows, each centered. No button was shrunk.
- **BorderLayout.** The top and bottom buttons stretched to the full width of 460 pixels at their preferred height of 25. The left and right buttons kept their preferred widths (187 and 120) and stretched to the full remaining height of 100. The center button received everything left: $460 - 187 - 120 = 153$ pixels wide.
- **BoxLayout.** One column; each button keeps its preferred size and is centered because we set `setAlignmentX(CENTER_ALIGNMENT)`.
- **GridBagLayout.** Column 0 is as wide as its widest label, *Weight (kg)*, and each label is anchored to the right end of its cell (`LINE_END`), which is why *Customer* starts at $x = 18$ and *Phone* at $x = 41$: they line up on the right. Column 1 has `weightx = 1` and `fill = HORIZONTAL`, so the text fields took all the remaining width (202 pixels) even though we asked for only 10 columns. The *Add order* button, with `fill = NONE`, kept its size and sat at the right end.

Listing 10.10 is the Tkinter tour, with `pack` and `grid`.

Listing 10.10 `layout_tour.py`: `pack` and `grid` at work; the self-test prints the geometry Tk assigns.

```
"""layout_tour.py: Tkinter's pack and grid geometry managers at work."""
import sys
import tkinter as tk

NAMES = ["Button 1", "Button 2", "Button 3", "Long-Named Button 4", "5"]


def row_of_buttons(master):  # pack side=LEFT: one row, like FlowLayout
    f = tk.Frame(master)
    for n in NAMES:
        tk.Button(f, text=n).pack(side=tk.LEFT, padx=2, pady=2)
    return f


def border_like(master):  # pack order decides who gets the edges
    f = tk.Frame(master)
    tk.Button(f, text="Button 1 (top)").pack(side=tk.TOP, fill=tk.X)
    tk.Button(f, text="Long-Named Button 4 (bottom)").pack(side=tk.BOTTOM, fill=tk.X)
    tk.Button(f, text="Button 3 (left)").pack(side=tk.LEFT, fill=tk.Y)
    tk.Button(f, text="5 (right)").pack(side=tk.RIGHT, fill=tk.Y)
    tk.Button(f, text="Button 2 (center)").pack(fill=tk.BOTH, expand=True)
    return f


def column_of_buttons(master):  # pack side=TOP: one column, like BoxLayout
    f = tk.Frame(master)
    for n in NAMES:
        tk.Button(f, text=n).pack(side=tk.TOP)
    return f


def form_grid(master):  # grid: rows and columns, like GridBagLayout
    f = tk.Frame(master)
    for row, text in enumerate(["Customer", "Phone", "Weight (kg)"]):
        tk.Label(f, text=text).grid(row=row, column=0, sticky="e", padx=4, pady=2)
        tk.Entry(f, width=10).grid(row=row, column=1, sticky="ew", padx=4, pady=2)
    tk.Button(f, text="Add order").grid(row=3, column=1, sticky="e", padx=4, pady=2)
    f.columnconfigure(1, weight=1)  # column 1 takes the extra width
    return f


def name(w):
    return w.cget("text") if w.winfo_class() != "Entry" else "[entry]"
```

Listing 10.10 (Continued) *layout_tour.py: pack and grid at work; the self-test prints the geometry Tk assigns.*

```
def self_test():
    root = tk.Tk()
    demos = [("pack LEFT", row_of_buttons, 560, 40),
             ("pack LEFT", row_of_buttons, 230, 100),
             ("pack edges", border_like, 460, 150),
             ("pack TOP", column_of_buttons, 230, 170),
             ("grid", form_grid, 300, 120)]
    frames = []
    for title, make, w, h in demos:
        holder = tk.Frame(root, width=w, height=h)
        holder.pack_propagate(False)  # keep the size we asked for
        holder.pack(anchor="w")
        inner = make(holder)
        inner.pack(fill=tk.BOTH, expand=True)
        frames.append((title, inner, w, h))
    root.geometry("+0+0")
    root.update()  # Tk computes geometry only for mapped windows
    for title, inner, w, h in frames:
        print(f"{title} in {w} x {h}:")
        for c in inner.winfo_children():
            print(f"  {name(c):31} x={c.winfo_x():3} y={c.winfo_y():3}"
                  f" w={c.winfo_width():3} h={c.winfo_height():3}")
    root.destroy()


def main():
    if "--selftest" in sys.argv:
        self_test()
        return
    root = tk.Tk()
    root.title("Layout tour")
    for make in (row_of_buttons, border_like, column_of_buttons, form_grid):
        make(root).pack(fill=tk.X, pady=6)
    root.mainloop()


if __name__ == "__main__":
    main()
```

```
pack LEFT in 560 x 40:
  Button 1                      x=  2 y=  4 w= 90 h= 31
  Button 2                      x= 96 y=  4 w= 90 h= 31
  Button 3                      x=190 y=  4 w= 90 h= 31
  Long-Named Button 4           x=284 y=  4 w=185 h= 31
  5                             x=473 y=  4 w= 39 h= 31
pack LEFT in 230 x 100:
  Button 1                      x=  2 y= 34 w= 90 h= 31
  Button 2                      x= 96 y= 34 w= 90 h= 31
  Button 3                      x=190 y= 34 w= 38 h= 31
  Long-Named Button 4           x=  0 y=  0 w=  1 h=  1
  5                             x=  0 y=  0 w=  1 h=  1
pack edges in 460 x 150:
  Button 1 (top)                x=  0 y=  0 w=460 h= 31
  Long-Named Button 4 (bottom)  x=  0 y=119 w=460 h= 31
  Button 3 (left)               x=  0 y= 31 w=127 h= 88
  5 (right)                     x=374 y= 31 w= 86 h= 88
  Button 2 (center)             x=127 y= 31 w=247 h= 88
pack TOP in 230 x 170:
  Button 1                      x= 70 y=  0 w= 90 h= 31
  Button 2                      x= 70 y= 31 w= 90 h= 31
  Button 3                      x= 70 y= 62 w= 90 h= 31
  Long-Named Button 4           x= 22 y= 93 w=185 h= 31
  5                             x= 95 y=124 w= 39 h= 31
grid in 300 x 120:
  Customer                      x= 16 y=  3 w= 73 h= 21
  [entry]                       x= 97 y=  2 w=199 h= 23
  Phone                         x= 41 y= 30 w= 48 h= 21
  [entry]                       x= 97 y= 29 w=199 h= 23
  Weight (kg)                   x=  4 y= 57 w= 85 h= 21
  [entry]                       x= 97 y= 56 w=199 h= 23
  Add order                     x=195 y= 83 w=101 h= 31
```

Compare the two outputs, because they teach the most important difference between the toolkits' simple layouts.

- **`pack(side=LEFT)` does not wrap.** In the 230-pixel frame, *Button 1* and *Button 2* got their full size, *Button 3* was squeezed to 38 pixels, and *Long-Named Button 4* and *5* got no space at all (a width and height of 1 is Tk's way of saying "not shown"). `pack` hands out space in the order the widgets were packed until it runs out; unlike `FlowLayout`, it never starts a new row. If you need wrapping in Tkinter, use `grid` and compute the rows yourself.
- **`pack` against the edges imitates `BorderLayout`**, but the *order* decides who owns the corners. We packed top and bottom first, so they span the whole width, and the left and right buttons only get the height between them — the same shape as **Figure 10.9**. Had we packed left and right first, they would have run the full height and the top and bottom would have been squeezed between them. The last widget, packed with `fill=BOTH, expand=True`, plays the role of `CENTER`.
- **`pack(side=TOP)`** is a column, centered by default, like the `BoxLayout`.
- **`grid`** reproduced the `GridBagLayout` form almost exactly: labels right-aligned with `sticky="e"`, entries stretched with `sticky="ew"` in a column of weight 1.

The Tk self-test must show its window for an instant (`root.update()` after `root.geometry("+0+0")`), because Tk computes real geometry only for windows mapped on a screen; on a server, `xvfb-run` provides an invisible one.

Table 10.3 puts the choices side by side.

Table 10.3 Swing layout managers and Tkinter geometry managers for common needs.

Need	Swing	Tkinter
A row that wraps, like words	`FlowLayout` (default for `JPanel`)	no direct equivalent; `grid` with computed rows
A single row or column	`BoxLayout` / `Box`	`pack(side=LEFT)` / `pack(side=TOP)`
Center plus four edges	`BorderLayout` (default for a content pane)	`pack` against the sides, edges first, then `fill=BOTH, expand=True`
A form: rows and columns, some stretching	`GridBagLayout` + `GridBagConstraints`	`grid(row, column, sticky)` + `columnconfigure(weight)`
Equal cells	`GridLayout(rows, cols)`	`grid` with `uniform` columns
Absolute positions (avoid)	`setLayout(null)` + `setBounds`	`place(x, y)`

✓ **KEY INSIGHT.** Real windows are built by *nesting*: a `BorderLayout` for the overall shape, a `GridBagLayout` (or `grid`) for the form in one region, a `FlowLayout` (or a packed frame) for a row of buttons inside the form. Each container solves one simple layout problem. When a layout seems impossible, the answer is almost always another level of panel, not a cleverer manager.

10.9 Painting: paintComponent and the Canvas

Components that come with the toolkit know how to draw themselves. When you want something that no component provides — a gray box that darkens, a chart, a plan of the laundry's machines — you draw it yourself. The two toolkits take opposite approaches here, and understanding both will make you better at either.

10.9.1 The coordinate system, and the Graphics object

A `JPanel` has a method

```
protected void paintComponent(Graphics g)
```

which a subclass overrides to paint arbitrary shapes. The parameter `g` is a `Graphics` object: a pen, already pointed at the panel, with a current color and font. Its drawing operations use a coordinate system that surprises everyone once: the origin (0,0) is the **top-left** corner of the panel, x grows to the right, and y grows **downward**, because screens are drawn line by line from the top. **Figure 10.12** shows four of the basic operations:

```
g.drawLine(x1, y1, x2, y2);      // a line from (x1, y1) to (x2, y2)
g.drawRect(x, y, xSize, ySize);  // the outline of a rectangle; (x, y) is its top-left
corner
g.fillRect(x, y, xSize, ySize);  // the same rectangle, filled
g.drawOval(x, y, xSize, ySize);  // the ellipse that fits in that rectangle
```

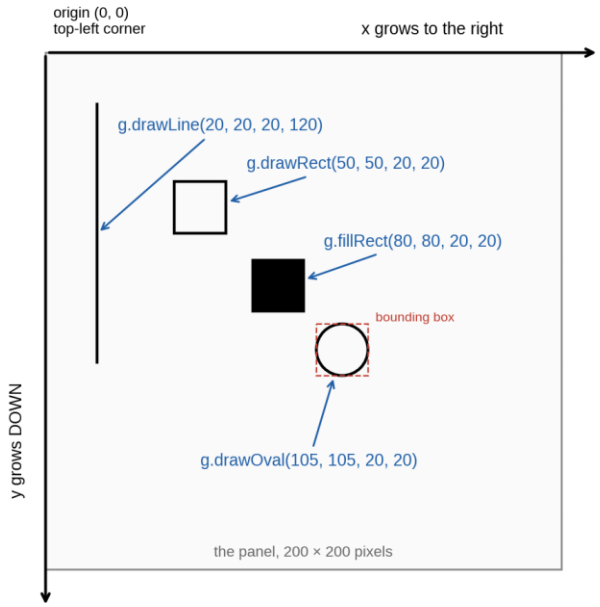


Figure 10.12 Swing's coordinate system: the origin is the top-left corner and y grows downward. An oval is specified by its bounding box, not by its center.

Notice that an oval is given by its *bounding box*: `drawOval(105, 105, 20, 20)` draws a circle of diameter 20 whose center is at (115,115). There are many more operations — `fillOval`, `drawString`, `drawPolygon`, `drawArc`, `drawImage` — and the subclass `Graphics2D`, which the `g` parameter really is in Swing, adds line widths, antialiasing, transformations and arbitrary shapes.

10.9.2 Color

Colors are expressed in the **RGB** model: an amount of red, green and blue light, each an integer from 0 to 255. The higher the number, the more of that light, so higher is brighter. `new Color(0, 0, 0)` is black (no light), `new Color(255, 255, 255)` is white (all three at full strength), and `new Color(255, 0, 0)` is bright red. When the three amounts are equal, the result is a shade of gray. The number of different colors this model can express is

$$256 \times 256 \times 256 = 256^3 = 16{,}777{,}216 \tag{10.1}$$

as Equation (10.1) computes: 256 choices for each of three independent channels. The class `Color` also has constants for common colors (`Color.RED`, `Color.BLUE`, `Color.WHITE`, …). `g.setColor(c)` sets the color for all *future* painting operations of that `Graphics` object, so it is called before the shapes it should affect. Tk writes the same colors as strings, either by name (`"blue"`) or as hexadecimal RGB, `"#rrggbb"`: `"#ff0000"` is bright red, `"#0a0a0a"` a very dark gray (10, 10, 10).

10.9.3 Who calls paintComponent?

You never call `paintComponent` yourself. The Java run-time system calls it whenever the panel needs to be painted: when the window first appears, when it is restored after being minimized, when another window that covered it moves away, when it is resized. Your method must therefore be able to paint the panel's *entire* current appearance, from scratch, at any moment — which means that everything it draws must come from the panel's fields, not from what happened to be drawn last time.

When your program changes the panel's state and wants the display to follow, it calls

```
void repaint()
```

which does not paint anything immediately. It marks the panel as "dirty" and returns [10]; the event loop later calls `paintComponent` (line 9 of **Pseudocode 10.1**). Several `repaint()` calls in quick succession are merged into one painting. This is why a listener that changes a field and calls `repaint()` never sees the effect during its own execution: the painting happens after the listener returns.

The first line of every `paintComponent` should be `super.paintComponent(g)`. It paints the panel's background (erasing the previous picture) and is mandatory: without it, old drawings remain visible under new ones, and on some systems random garbage appears.

10.9.4 A box that darkens and lightens

Listing 10.11 combines painting, action events, mouse events and `getSource()`. A panel draws a gray box. Two buttons, *Darken* and *Lighten*, make it 10 units darker or lighter; clicking the box itself also lightens it. The gray level cannot go below 0 or above 255, and the button that would push it past a limit is disabled.

Listing 10.11 `ShadeBox.java`: painting in `paintComponent`, two buttons told apart with `getSource()`, and mouse clicks on the painted box.

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Rectangle;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.image.BufferedImage;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class ShadeBox extends JPanel implements ActionListener, MouseListener {
    static final Rectangle BOX = new Rectangle(10, 40, 100, 50);
    private final JButton darken = new JButton("Darken");
    private final JButton lighten = new JButton("Lighten");
    private int level = 255;  // 255 = white, 0 = black

    public ShadeBox() {
        setPreferredSize(new Dimension(300, 150));
        add(darken);
        add(lighten);
        darken.addActionListener(this);
        lighten.addActionListener(this);
        addMouseListener(this);  // the panel itself produces mouse events
        lighten.setEnabled(false);
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);  // mandatory: paint the background first
        g.setColor(new Color(level, level, level));
        g.fillRect(BOX.x, BOX.y, BOX.width, BOX.height);
        g.setColor(Color.BLACK);
        g.drawRect(BOX.x, BOX.y, BOX.width, BOX.height);
    }

    static int clamp(int x) {
        return Math.max(0, Math.min(255, x));
    }
```

Listing 10.11 (Continued) *ShadeBox.java: painting in paintComponent, two buttons told apart with getSource(), and mouse clicks on the painted box.*

```
    private void shade(int delta) {
        level = clamp(level + delta);
        darken.setEnabled(level > 0);
        lighten.setEnabled(level < 255);
        repaint();  // ask for paintComponent to run soon
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == darken) {  // which button was it?
            shade(-10);
        } else if (e.getSource() == lighten) {
            shade(+10);
        }
    }

    @Override
    public void mouseClicked(MouseEvent e) {
        if (BOX.contains(e.getPoint())) {
            shade(+10);
        }
    }

    @Override public void mousePressed(MouseEvent e) { }
    @Override public void mouseReleased(MouseEvent e) { }
    @Override public void mouseEntered(MouseEvent e) { }
    @Override public void mouseExited(MouseEvent e) { }

    public static void main(String[] args) throws Exception {
        if (args.length > 0 && args[0].equals("--selftest")) {
            System.setProperty("java.awt.headless", "true");
            SwingUtilities.invokeAndWait(ShadeBox::selfTest);
        } else {
            SwingUtilities.invokeLater(() -> {
                JFrame.setDefaultLookAndFeelDecorated(true);
                JFrame frame = new JFrame("Shade box");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.add(new ShadeBox());
                frame.pack();
                frame.setVisible(true);
            });
        }
    }
```

Listing 10.11 (Continued) *ShadeBox.java: painting in paintComponent, two buttons told apart with getSource(), and mouse clicks on the painted box.*

```
    String state() {
        return "level=" + level + " darken=" + (darken.isEnabled() ? "on" : "off")
                + " lighten=" + (lighten.isEnabled() ? "on" : "off");
    }

    void click(int x, int y) {  // deliver a mouse click exactly as AWT would
        dispatchEvent(new MouseEvent(this, MouseEvent.MOUSE_CLICKED,
                System.currentTimeMillis(), 0, x, y, 1, false, MouseEvent.BUTTON1));
    }

    static void selfTest() {
        ShadeBox p = new ShadeBox();
        System.out.println("start:           " + p.state());
        p.darken.doClick();
        System.out.println("darken x1:       " + p.state());
        for (int i = 0; i < 30; i++) {
            p.darken.doClick();  // a disabled button ignores the click
        }
        System.out.println("darken x30 more: " + p.state());
        p.click(60, 65);
        System.out.println("click in box:    " + p.state());
        p.click(250, 130);
        System.out.println("click outside:   " + p.state());
        p.setSize(300, 150);
        BufferedImage img = new BufferedImage(300, 150, BufferedImage.TYPE_INT_RGB);
        p.paint(img.getGraphics());  // run the painting code into an image
        System.out.printf("pixel in box:    #%06x%n", img.getRGB(60, 65) & 0xFFFFFF);
        System.out.printf("pixel outside:   #%06x%n", img.getRGB(250, 130) & 0xFFFFFF);
    }
}
```

The class declares that it listens to two types of events: `implements ActionListener, MouseListener`. The constructor registers the panel with both buttons (`addActionListener(this)`) and with itself for mouse events (`addMouseListener(this)` — the panel is the component the mouse clicks). There is a single `actionPerformed` for both buttons, so it uses `e.getSource() == darken` to tell them apart; `==` is correct here, because we are asking whether the source is *the very same object* as our `darken` button (Chapter 3's reference equality).

The heart of the class is `shade(delta)`. It changes the state, `level`, and keeps it in range with `clamp`, which computes

$$v' = \min(255, \max(0,\ v + \delta)) \tag{10.2}$$

where v is the current level and δ the change, -10 or $+10$. Equation (10.2) reads from the inside out: add the change, raise the result to 0 if it went negative, lower it to 255 if it went above. Then `shade` enables *Darken* only while the box is not yet black and *Lighten* only while it is not yet white, and calls `repaint()`. It does not draw. The drawing happens in `paintComponent`, which reads `level` and paints the box accordingly, whenever Swing decides to call it.

`MouseListener` has five methods, and a class that implements the interface must define all five, even though we only need `mouseClicked`. The other four have empty bodies. This is a nuisance that Section 10.10 removes with an *adapter class*. The click handler asks `BOX.contains(e.getPoint())` — is the clicked point inside the box's rectangle? — so clicks elsewhere on the panel are ignored.

How many clicks on *Darken* does it take to go from white to black? Each click subtracts 10, and the last click is clamped, so the answer is

$$\lceil \frac{255}{10} \rceil = \lceil 25.5 \rceil = 26 \tag{10.3}$$

by Equation (10.3): after 25 clicks the level is $255 - 250 = 5$, and the 26th click would give -5, which Equation (10.2) clamps to 0. The self-test checks exactly this, and then checks the painting itself: it gives the panel a size, calls `paint` on an in-memory `BufferedImage` instead of the screen — the same painting code, drawing into an image — and reads back the color of two pixels.

```
start:           level=255 darken=on lighten=off
darken x1:       level=245 darken=on lighten=on
darken x30 more: level=0 darken=off lighten=on
click in box:    level=10 darken=on lighten=on
click outside:   level=10 darken=on lighten=on
pixel in box:    #0a0a0a
pixel outside:   #eeeeee
```

After one click the level is 245 and both buttons are enabled. The loop then clicks 30 more times; the level reaches 0 after the 25th of them (the 26th click overall), *Darken* is disabled, and the remaining clicks on a disabled button do nothing — `doClick()` on a disabled button does not fire. A click at (60,65), inside the box, lightens it to 10; a click at (250,130), outside, changes nothing. And the pixel inside the box is `#0a0a0a`, that is, red, green and blue all equal to $0a_{16} = 10$: the painting code really painted what the state says. The pixel outside is the panel's default background, a light gray `#eeeeee`. **Figure 10.13** shows the same program on screen at the moment the box has become black and *Darken* has been disabled.



Figure 10.13 The shading window after enough clicks on *Darken*: the box is black, and the *Darken* button is disabled because the level cannot go lower.

A variant, in **Figure 10.14**, darkens with a button, lightens with a mouse click, and explains both in a label that changes at the limits — one `setText` that saves the user from wondering why a button stopped working.



Figure 10.14 A variant that darkens with a button and lightens with a mouse click, and says so in a label; here the box has reached black and the label reports that it cannot be darkened further.

10.9.5 The Tkinter Canvas: shapes that remember themselves

Listing 10.12 is the same box in Tkinter.

Listing 10.12 `shade_box.py`: the shading box on a Tkinter `Canvas`, where the box is an item that Tk remembers and redraws.

```
"""shade_box.py: darken and lighten a gray box drawn on a Tkinter canvas."""
import sys
import tkinter as tk


def clamp(x):
    return max(0, min(255, x))


class ShadeBox(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.level = 255  # 255 = white, 0 = black
        bar = tk.Frame(self)
        bar.pack(side=tk.TOP, pady=4)
        self.darken = tk.Button(bar, text="Darken", command=lambda: self.shade(-10))
        self.lighten = tk.Button(bar, text="Lighten", command=lambda: self.shade(+10),
                                 state=tk.DISABLED)
        self.darken.pack(side=tk.LEFT, padx=2)
        self.lighten.pack(side=tk.LEFT, padx=2)
        self.canvas = tk.Canvas(self, width=300, height=110, highlightthickness=0)
        self.canvas.pack()
        # A canvas item is an object that stays on the canvas until deleted.
        self.box = self.canvas.create_rectangle(10, 10, 110, 60,
                                                fill=self.color(), outline="black")
        self.canvas.tag_bind(self.box, "<Button-1>", self.on_box_click)

    def color(self):
        return "#%02x%02x%02x" % (self.level, self.level, self.level)

    def shade(self, delta):
        self.level = clamp(self.level + delta)
        self.darken.config(state=tk.NORMAL if self.level > 0 else tk.DISABLED)
        self.lighten.config(state=tk.NORMAL if self.level < 255 else tk.DISABLED)
        self.canvas.itemconfig(self.box, fill=self.color())  # Tk redraws by itself

    def on_box_click(self, event):  # Tk calls this only for clicks on the box
        self.shade(+10)

    def state(self):
        return (f"level={self.level}"
                f" darken={'on' if self.darken['state'] == tk.NORMAL else 'off'}"
                f" lighten={'on' if self.lighten['state'] == tk.NORMAL else 'off'}")
```

Listing 10.12 (Continued) *shade_box.py: the shading box on a Tkinter Canvas, where the box is an item that Tk remembers and redraws.*

```
def self_test():
    root = tk.Tk()
    root.withdraw()
    p = ShadeBox(root)
    p.pack()
    print("start:          ", p.state())
    p.darken.invoke()
    print("darken x1:      ", p.state())
    for _ in range(30):
        p.darken.invoke()  # a disabled button ignores invoke()
    print("darken x30 more:", p.state())
    p.on_box_click(None)  # what Tk calls for a click on the box
    print("click in box:   ", p.state())
    print("box fill:       ", p.canvas.itemcget(p.box, "fill"))
    print("canvas items:   ", p.canvas.find_all())
    root.destroy()


def main():
    if "--selftest" in sys.argv:
        self_test()
        return
    root = tk.Tk()
    root.title("Shade box")
    ShadeBox(root).pack()
    root.mainloop()


if __name__ == "__main__":
    main()
```

```
start:           level=255 darken=on lighten=off
darken x1:       level=245 darken=on lighten=on
darken x30 more: level=0 darken=off lighten=on
click in box:    level=10 darken=on lighten=on
box fill:        #0a0a0a
canvas items:    (1,)
```

There is no `paintComponent` and no `repaint()`. A Tkinter `Canvas` works in **retained mode**: `create_rectangle(10, 10, 110, 60, fill=...)` does not draw pixels and forget them, as `fillRect` does; it creates an *item*, an object that lives on the canvas and has an identifier (here 1, as the last line of the output shows). Tk remembers all the items and redraws them itself whenever the window needs it. To change the picture you change an item — `itemconfig(self.box, fill=...)` — or move it (`move`, `coords`) or delete it (`delete`), and Tk updates the screen on its own. Swing's approach is called **immediate mode**: the panel keeps no picture, only your state, and `paintComponent` turns the state into pixels every time.

Retained mode brings a second gift: items can receive events. `canvas.tag_bind(self.box, "<Button-1>", self.on_box_click)` binds the callback to the *box item*, not to the whole canvas, so Tk itself decides whether the click was on the box — no `contains` test is needed. The self-test calls `on_box_click(None)` for the reason given in Section 10.7.1, and uses `itemcget(self.box, "fill")` to read back the color, the retained-mode counterpart of reading a pixel.

> ✓ **KEY INSIGHT.** In Swing (immediate mode) the rule is: *state in fields, pixels in `paintComponent`, and `repaint()` to connect them*. In a Tk canvas (retained mode) the rule is: *create items once, then change the items*. Both keep the picture a faithful function of the program's state; they differ only in who stores the picture.

10.10 Inner Classes, Anonymous Classes, and Lambdas

So far our panels have been their own listeners: `class ShadeBox extends JPanel implements ActionListener, MouseListener`. That works, but it has three drawbacks that grow with the window. First, one `actionPerformed` must serve every button, so it fills up with `if (e.getSource() == …)` tests. Second, the class must define all five `MouseListener` methods even if it needs one. Third, the panel's public interface now includes `actionPerformed` and `mouseClicked`, methods that no other class should ever call. The laundry window has seven different reactions to seven different events; one method with seven branches would be unreadable.

Java's answer, since JDK 1.1, is the **inner class**; Java 8 added **lambda expressions** as a shorter form for the most common case. Python's answer is that functions are objects and can be defined anywhere, including inside other functions, where they become **closures**.

10.10.1 Inner classes

> **DEFINITION.** An **inner class** is a class declared inside the body of another class (the *outer* class) without the modifier `static`. Every object of an inner class is created *inside* an object of the outer class and keeps a hidden reference to it, so the inner object can use the outer object's fields and methods — even `private` ones — as if they were its own. An inner object's reference to its outer object can be written explicitly as `Outer.this`.

Inner classes are often used to define listeners for events. Instead of one `actionPerformed` that asks which button was clicked, we write one small class per reaction:

```
private class DarkenListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        shade(-10);           // the outer panel's method
    }
}
...
darken.addActionListener(new DarkenListener());
```

The inner class can access the instance variables of the outer class (`level`, `darken`, `lighten`) as well as its methods (`shade`, `repaint()`), and an instance of it can be supplied as the listener. Each event is now handled separately in a self-contained method — `DarkenListener.actionPerformed`, `LightenListener.actionPerformed` — which leads to better structure when a class has to listen to many events.

10.10.2 Anonymous classes and lambdas

Often an inner class is used exactly once, right where it is declared. Java lets you declare and instantiate it in one expression, without a name, as an **anonymous class**:

```
darken.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        shade(-10);
    }
});
```

Read `new ActionListener() { … }` as "create an object of a new, nameless class that implements `ActionListener` with this body". An anonymous class can also *extend* a class, which solves the five-methods problem: `java.awt.event.MouseAdapter` is a class that implements `MouseListener`, `MouseMotionListener` and `MouseWheelListener` with empty methods, so an anonymous subclass of it overrides only the methods it needs. Such do-nothing implementations of listener interfaces

are called **adapter classes**; AWT provides one for every listener interface with more than one method (`MouseAdapter`, `KeyAdapter`, `WindowAdapter`, …).

An interface with exactly one abstract method, such as `ActionListener`, is a **functional interface**, and Java 8 introduced a still shorter notation for implementing one: the **lambda expression**

```
darken.addActionListener(e -> shade(-10));
```

which says: "an `ActionListener` whose `actionPerformed(e)` does `shade(-10)`". The compiler knows, from the parameter type of `addActionListener`, which interface and which method are meant. When the body only calls an existing method with the same parameters, a **method reference** is shorter still: `reset.addActionListener(this::resetCount)` passes the method `resetCount(ActionEvent e)` of this object as the listener [11].

10.10.3 Five styles side by side

Listing 10.13 wires five buttons in the five styles — the panel itself, a named inner class, an anonymous class, a lambda and a method reference — and three more buttons with lambdas created in a loop.

Listing 10.13 `ListenerStyles.java`: five ways to write an action listener in Java, and lambdas created in a loop.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.List;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class ListenerStyles extends JPanel implements ActionListener {
    private int count = 0;
    private final JLabel shown = new JLabel("count = 0");
    private final JButton plusOne = new JButton("+1");     // style 1: the panel itself
    private final JButton plusTwo = new JButton("+2");     // style 2: named inner class
    private final JButton plusThree = new JButton("+3");   // style 3: anonymous class
    private final JButton plusFour = new JButton("+4");    // style 4: lambda
    private final JButton reset = new JButton("Reset");    // style 5: method reference
    private final List<JButton> loopButtons = new ArrayList<>();

    public ListenerStyles() {
        for (JButton b : List.of(plusOne, plusTwo, plusThree, plusFour, reset)) {
            add(b);
        }
        plusOne.addActionListener(this);
        plusTwo.addActionListener(new AddListener(2));
        plusThree.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                bump(3);
            }
        });
        int step = 4;  // effectively final, so a lambda may capture it
        plusFour.addActionListener(e -> bump(step));
        reset.addActionListener(this::resetCount);
        for (int n : new int[] {10, 20, 30}) {  // a fresh n for every pass
            JButton b = new JButton("+" + n);
            b.addActionListener(e -> bump(n));
            loopButtons.add(b);
            add(b);
        }
        add(shown);
    }
```

Listing 10.13 (Continued) *ListenerStyles.java: five ways to write an action listener in Java, and lambdas created in a loop.*

```
    private void bump(int n) {
        count += n;
        shown.setText("count = " + count);
    }

    private void resetCount(ActionEvent e) {
        count = 0;
        shown.setText("count = " + count);
    }

    @Override
    public void actionPerformed(ActionEvent e) {  // style 1
        bump(1);
    }

    private class AddListener implements ActionListener {  // style 2
        private final int n;

        AddListener(int n) {
            this.n = n;
        }

        @Override
        public void actionPerformed(ActionEvent e) {
            bump(n);  // an inner object may call its outer object's methods
            System.out.println("  AddListener sees ListenerStyles.this.count = "
                    + ListenerStyles.this.count);
        }
    }

    public static void main(String[] args) throws Exception {
        if (args.length > 0 && args[0].equals("--selftest")) {
            System.setProperty("java.awt.headless", "true");
            SwingUtilities.invokeAndWait(ListenerStyles::selfTest);
        } else {
            SwingUtilities.invokeLater(() -> {
                JFrame.setDefaultLookAndFeelDecorated(true);
                JFrame frame = new JFrame("Listener styles");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.add(new ListenerStyles());
                frame.pack();
                frame.setVisible(true);
            });
        }
    }
```

Listing 10.13 (Continued) *ListenerStyles.java: five ways to write an action listener in Java, and lambdas created in a loop.*

```
    static void selfTest() {
        ListenerStyles p = new ListenerStyles();
        for (JButton b : List.of(p.plusOne, p.plusTwo, p.plusThree, p.plusFour)) {
            b.doClick();
            System.out.println(b.getText() + " -> " + p.shown.getText());
        }
        for (JButton b : p.loopButtons) {
            b.doClick();
            System.out.println(b.getText() + " -> " + p.shown.getText());
        }
        p.reset.doClick();
        System.out.println("Reset -> " + p.shown.getText());
        System.out.println("listener classes:");
        System.out.println("  " + p.plusTwo.getActionListeners()[0].getClass().getName());
        System.out.println("  " +
p.plusThree.getActionListeners()[0].getClass().getName());
        System.out.println("  lambda class is hidden: "
                + p.plusFour.getActionListeners()[0].getClass().isHidden());
    }
}
```

```
+1 -> count = 1
  AddListener sees ListenerStyles.this.count = 3
+2 -> count = 3
+3 -> count = 6
+4 -> count = 10
+10 -> count = 20
+20 -> count = 40
+30 -> count = 70
Reset -> count = 0
listener classes:
  ListenerStyles$AddListener
  ListenerStyles$1
  lambda class is hidden: true
```

The output confirms that all five styles do the same job: $1,1 + 2 = 3,3 + 3 = 6,6 + 4 = 10$, then $10 + 10 = 20$, $20 + 20 = 40$, $40 + 30 = 70$, and *Reset* returns to 0. The line printed from inside `AddListener` shows that the inner object reads the outer object's `count` through `ListenerStyles.this.count` — at that moment already 3, because `bump(n)` ran first.

The last three lines show what the compiler made of our listeners. The named inner class is a real class called `ListenerStyles$AddListener`; the anonymous class got the name `ListenerStyles$1` (the first anonymous class in `ListenerStyles`). Compiling the file indeed produces three class files:

```
ListenerStyles$1.class
ListenerStyles$AddListener.class
ListenerStyles.class
```

The lambda produced no class file: the JVM generates a *hidden class* for it at run time, when the lambda is first used, which is why `isHidden()` is `true` and why its name (something like `ListenerStyles$$Lambda/0x…`) changes from run to run.

Figure 10.15 shows the objects behind the listeners. The inner and the anonymous listener each hold a hidden field, which the compiler names `this$0`, pointing to the panel that created them; that is how `bump(n)` inside them reaches the panel's `count`. The lambda object holds the values it *captured*: a reference to the panel (because `bump` is an instance method) and a *copy* of `step`.

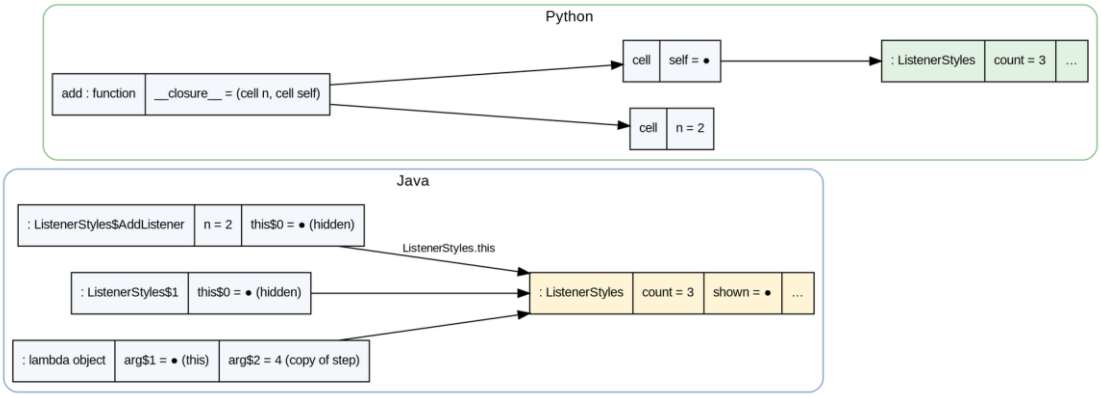


Figure 10.15 What listeners remember. Java inner and anonymous objects keep a hidden reference to their outer object; a lambda object holds copies of the values it captured. A Python closure keeps *cells* that refer to the variables of the function that created it. (The field names are the compilers' implementation details.)

10.10.4 Effectively final: why Java copies

Look again at `int step = 4;` and `e -> bump(step)`. A lambda (or an anonymous or local class) may use a local variable of the enclosing method only if that variable is **effectively final** — assigned once and never changed. The reason is the copy in **Figure 10.15**. The lambda may be called long after the method that created it has returned and its local variables have disappeared from the stack; so Java stores a *copy* of the value inside the lambda object. If the variable could change afterwards, the copy and the variable would disagree, and the program would be confusing. Java rules the confusion out at compile time [11]. Here is what happens if we try the natural loop:

```
for (int n = 10; n <= 30; n += 10) {
    JButton b = new JButton("+" + n);
    b.addActionListener(e -> count += n);
}
```

```
LoopCapture.java:9: error: local variables referenced from a lambda expression must be
final or effectively final
            b.addActionListener(e -> count += n);
                                              ^
1 error
```

The variable `n` changes (`n += 10`), so it cannot be captured. The enhanced `for` loop in **Listing 10.13**, `for (int n : new int[] {10, 20, 30})`, is fine, because it declares a *new* variable `n` for every pass, and each one is assigned only once. Fields are a different matter: a lambda can read and change `count` freely, because it reaches `count` through the captured reference to the panel, not through a copy. That is also why the toy event loop of Section 10.6 used a one-element array: the *array reference* is effectively final, while its contents may change.

10.10.5 Python: bound methods, closures, lambdas, partial, callable objects

Python has no listener interfaces, so anything callable can be a callback. **Listing 10.14** shows five kinds, and the one trap they share.

Listing 10.14 `listener_styles.py`: five kinds of Tkinter callback, the late-binding trap in a loop, and its cure.

```
"""listener_styles.py: five ways to give a Tkinter button its callback."""
import sys
import tkinter as tk
from functools import partial


class Resetter:  # style 5: an object that can be called like a function
    def __init__(self, panel):
        self.panel = panel

    def __call__(self):
        self.panel.count = 0
        self.panel.show()


class ListenerStyles(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.count = 0
        self.shown = tk.Label(self, text="count = 0")

        def make_adder(n):  # style 2: a nested function that remembers n
            def add():
                self.bump(n)
            return add

        self.buttons = [
            tk.Button(self, text="+1", command=self.plus_one),        # 1: bound method
            tk.Button(self, text="+2", command=make_adder(2)),        # 2: closure
            tk.Button(self, text="+3", command=lambda: self.bump(3)),  # 3: lambda
            tk.Button(self, text="+4", command=partial(self.bump, 4)),  # 4: partial
        ]
        self.reset = tk.Button(self, text="Reset", command=Resetter(self))
        self.adder_two = make_adder(2)  # kept only so the self-test can inspect it
        self.late_first = None
        self.late = []   # the late-binding trap
        self.fixed = []  # the cure: a default argument freezes n now
        for n in (10, 20, 30):
            wrong = lambda: self.bump(n)  # looks n up when CALLED, not now
            self.late_first = self.late_first or wrong
            self.late.append(tk.Button(self, text=f"+{n}", command=wrong))
            self.fixed.append(tk.Button(self, text=f"+{n}", command=lambda n=n:
self.bump(n)))
        for b in self.buttons + [self.reset] + self.fixed:
            b.pack(side=tk.LEFT)
        self.shown.pack(side=tk.LEFT, padx=6)
```

Listing 10.14 (Continued) *listener_styles.py: five kinds of Tkinter callback, the late-binding trap in a loop, and its cure.*

```
    def plus_one(self):
        self.bump(1)

    def bump(self, n):
        self.count += n
        self.show()

    def show(self):
        self.shown.config(text=f"count = {self.count}")


def self_test():
    root = tk.Tk()
    root.withdraw()
    p = ListenerStyles(root)
    for b in p.buttons:
        b.invoke()
        print(b.cget("text"), "->", p.shown.cget("text"))
    p.reset.invoke()
    print("Reset ->", p.shown.cget("text"))
    for label, row in (("late", p.late), ("fixed", p.fixed)):
        for b in row:
            b.invoke()
            print(f"{label:5} {b.cget('text')} -> {p.shown.cget('text')}")
        p.reset.invoke()
    code = p.adder_two.__code__
    cells = [c.cell_contents for c in p.adder_two.__closure__]
    print("make_adder(2) remembers", code.co_freevars, "=", cells[0], "and the panel")
    print("first late lambda sees n =", p.late_first.__closure__[0].cell_contents)
    root.destroy()


def main():
    if "--selftest" in sys.argv:
        self_test()
        return
    root = tk.Tk()
    root.title("Listener styles")
    ListenerStyles(root).pack(padx=6, pady=6)
    root.mainloop()


if __name__ == "__main__":
    main()
```

```
+1 -> count = 1
+2 -> count = 3
+3 -> count = 6
+4 -> count = 10
Reset -> count = 0
late  +10 -> count = 30
late  +20 -> count = 60
late  +30 -> count = 90
fixed +10 -> count = 10
fixed +20 -> count = 30
fixed +30 -> count = 60
make_adder(2) remembers ('n', 'self') = 2 and the panel
first late lambda sees n = 30
```

1. **A bound method**, `self.plus_one`, is the method together with the object it belongs to. Tk calls it with no arguments (a `command` callback receives none), and Python supplies `self`. This is the most common style and the closest to Java's `this::method`.

2. **A closure.** `make_adder(2)` returns the nested function `add`, which uses `n` and `self` from the function that created it. A function that remembers variables of an enclosing function after that function has returned is a **closure** [12], and the last lines of the output inspect it: `add.__code__.co_freevars` lists the remembered names, `('n', 'self')`, and `add.__closure__` holds a **cell** for each, the first containing 2. **Figure 10.15** draws those cells.
3. **A lambda**, `lambda: self.bump(3)`, is an anonymous function of one expression — Python's counterpart of Java's lambda. It is a closure too; it remembers `self`.
4. `functools.partial(self.bump, 4)` builds a new callable that, when called, calls `self.bump(4)`: the argument is fixed now, the call happens later.
5. **A callable object.** `Resetter` has a `__call__` method, so its instances can be called like functions. This is the closest Python comes to Java's named listener class, with one difference: a Python class nested in another has no hidden reference to an outer object, so `Resetter` receives the panel explicitly in its constructor.

Now the trap. The first loop creates three buttons with `lambda: self.bump(n)`, and all three add 30. The reason is the difference between *Java's copies* and *Python's cells*. A Python closure does not store the value of `n` when it is created; it stores a reference to the *variable* `n`, and looks up the variable's value when the lambda is *called*. By the time any button is clicked, the loop has finished and `n` is 30, so all three lambdas see 30 — the output's last line confirms that the first lambda "sees" 30. This is Python's famous **late-binding** behavior [13]. Java prevents the same mistake by refusing to compile it; Python allows it and surprises you at run time.

The cure is the second loop: `lambda n=n: self.bump(n)`. A default argument is evaluated *once, when the lambda is created*, so each lambda gets its own parameter `n` with the current value as its default. `partial(self.bump, n)` would work equally well, and so would calling a factory such as `make_adder(n)`, which creates a new cell for every call. **Table 10.4** summarizes the choices.

Table 10.4 Ways to write a listener or callback, and what each one remembers.

Style	Java	Python	Remembers	Best for
The component implements the listener	`class P extends JPanel implements ActionListener`	—	itself	tiny programs only
Named inner class	`private class DarkenListener implements ActionListener`	callable object (`__call__`)	outer object (Java: hidden `this$0`)	listeners with their own state or several methods
Anonymous class	`new ActionListener() { … }`, `new MouseAdapter() { … }`	—	outer object, effectively final locals	multi-method listeners and adapters
Lambda	`e -> shade(-10)`	`lambda: self.shade(-10)`	Java: copies of effectively final locals; Python: cells, read at call time	one-line reactions
Method reference / bound method	`this::resetCount`	`self.reset_count`	the object	a reaction that deserves a name
Fixed arguments	lambda with a captured constant	`functools.partial(f, 4)`, `lambda n=n: f(n)`	the values given	many similar buttons in a loop

The modern Java guideline is to prefer lambdas and method references to anonymous classes whenever the interface has a single method, and anonymous classes (usually of an adapter) when it has several [14]. The laundry window follows that guideline, and so does the next program.

10.10.6 Drawing with the mouse

The mouse motion events of **Table 10.2** report every movement of the pointer (`mouseMoved`) and every movement with a button held down (`mouseDragged`). **Listing 10.15** turns them into a drawing program. It uses one anonymous `MouseAdapter` as both the mouse listener (for presses) and the mouse-motion listener (for drags and moves).

Listing 10.15 `Doodle.java`: drawing with the mouse, with an anonymous `MouseAdapter` and a painting method that redraws every stored stroke.

```
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Point;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.image.BufferedImage;
import java.util.ArrayList;
import java.util.List;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class Doodle extends JPanel {
    private final List<List<Point>> strokes = new ArrayList<>();  // the drawing
    private final JLabel xy = new JLabel("Use your mouse to draw a picture.");

    public Doodle() {
        setPreferredSize(new Dimension(300, 300));
        add(xy);
        MouseAdapter pen = new MouseAdapter() {  // an anonymous inner class
            @Override
            public void mousePressed(MouseEvent e) {
                strokes.add(new ArrayList<>(List.of(e.getPoint())));  // a new stroke
                repaint();
            }

            @Override
            public void mouseDragged(MouseEvent e) {
                strokes.get(strokes.size() - 1).add(e.getPoint());
                report(e);
                repaint();
            }

            @Override
            public void mouseMoved(MouseEvent e) {
                report(e);
            }
        };
        addMouseListener(pen);        // presses, releases, clicks
        addMouseMotionListener(pen);  // moves and drags
    }
```

Listing 10.15 (Continued) *Doodle.java: drawing with the mouse, with an anonymous MouseAdapter and a painting method that redraws every stored stroke.*

```
    private void report(MouseEvent e) {
        xy.setText("X: " + e.getX() + ", Y: " + e.getY());
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        g2.setColor(Color.BLUE);
        g2.setStroke(new BasicStroke(12, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND));
        for (List<Point> s : strokes) {  // redraw EVERYTHING from the stored points
            Point a = s.get(0);
            g2.drawLine(a.x, a.y, a.x, a.y);  // a dot, in case the stroke is one point
            for (Point b : s) {
                g2.drawLine(a.x, a.y, b.x, b.y);
                a = b;
            }
        }
    }

    public static void main(String[] args) throws Exception {
        if (args.length > 0 && args[0].equals("--selftest")) {
            System.setProperty("java.awt.headless", "true");
            SwingUtilities.invokeAndWait(Doodle::selfTest);
        } else {
            SwingUtilities.invokeLater(() -> {
                JFrame.setDefaultLookAndFeelDecorated(true);
                JFrame frame = new JFrame("Mouse motion event");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.add(new Doodle());
                frame.pack();
                frame.setVisible(true);
            });
        }
    }

    void send(int id, int x, int y) {
        dispatchEvent(new MouseEvent(this, id, System.currentTimeMillis(),
                MouseEvent.BUTTON1_DOWN_MASK, x, y, 1, false, MouseEvent.BUTTON1));
    }
```

Listing 10.15 (Continued) *Doodle.java: drawing with the mouse, with an anonymous MouseAdapter and a painting method that redraws every stored stroke.*

```
    static void selfTest() {
        Doodle d = new Doodle();
        d.send(MouseEvent.MOUSE_PRESSED, 50, 100);
        d.send(MouseEvent.MOUSE_DRAGGED, 100, 100);
        d.send(MouseEvent.MOUSE_DRAGGED, 150, 150);
        d.send(MouseEvent.MOUSE_RELEASED, 150, 150);
        d.send(MouseEvent.MOUSE_MOVED, 200, 60);
        System.out.println("strokes: " + d.strokes.size()
                + ", points in stroke 1: " + d.strokes.get(0).size());
        System.out.println("label:   " + d.xy.getText());
        d.setSize(300, 300);
        BufferedImage img = new BufferedImage(300, 300, BufferedImage.TYPE_INT_RGB);
        d.paint(img.getGraphics());
        System.out.printf("pixel at (75, 100):  #%06x%n", img.getRGB(75, 100) & 0xFFFFFF);
        System.out.printf("pixel at (250, 250): #%06x%n", img.getRGB(250, 250) & 0xFFFFFF);
    }
}
```

```
strokes: 1, points in stroke 1: 3
label:   X: 200, Y: 60
pixel at (75, 100):  #0000ff
pixel at (250, 250): #eeeeee
```

The panel's state is `strokes`, a `List<List<Point>>`. Pressing starts a new stroke, dragging adds a point and calls `repaint()`, and moving reports the coordinates. `paintComponent` redraws *every* stroke from the stored points with a thick, round-ended blue pen. The self-test sends real `MouseEvent`s with `dispatchEvent`, exactly as AWT delivers them — a press at (50,100), drags to (100,100) and (150,150), a release, a move to (200,60) — then paints into an image and finds blue halfway along the first segment and plain background far away.

Why store the points at all? Why not draw each segment directly in `mouseDragged`, with `Graphics g = getGraphics(); g.fillOval(...)`? That shortcut appears in many tutorials, and it seems to work: you can draw a heart like the one in **Figure 10.16**.

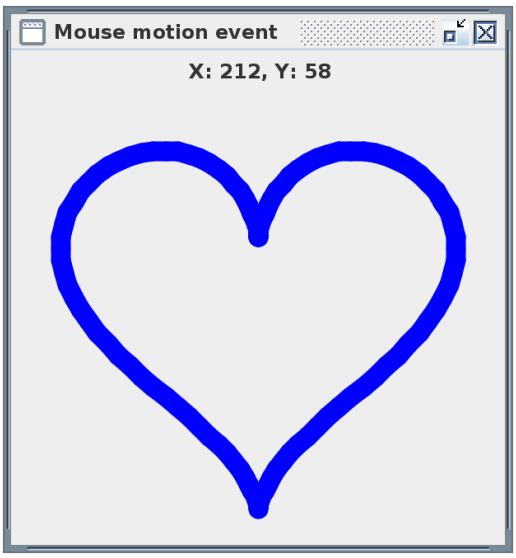


Figure 10.16 A heart drawn by dragging the mouse, with the coordinates of the pointer shown in the label.

But minimize the window and restore it, or cover it with another window, and the heart is gone. Painting done outside `paintComponent` is not remembered anywhere; the next time Swing repaints the panel — which it may do at any moment, for reasons of its own — it calls `paintComponent`, which knows nothing about the heart. Storing the points in a field and drawing them in `paintComponent` is what makes the drawing survive. (For a panel that is not on a screen, `getGraphics()` even returns `null`, so the shortcut version cannot be tested without a display at all.)

Listing 10.16 does the same in Tkinter, where retained mode makes the problem disappear by itself: each segment becomes a canvas line item, and Tk redraws the items whenever needed.

Listing 10.16 `doodle.py`: drawing with the mouse on a Tkinter canvas; every segment is a line item that Tk keeps.

```
"""doodle.py: draw with the mouse on a Tkinter canvas."""
import sys
import tkinter as tk
from types import SimpleNamespace


class Doodle(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.xy = tk.Label(self, text="Use your mouse to draw a picture.")
        self.xy.pack()
        self.canvas = tk.Canvas(self, width=300, height=280, bg="white")
        self.canvas.pack()
        self.last = None  # the previous point of the current stroke
        self.canvas.bind("<ButtonPress-1>", self.on_press)
        self.canvas.bind("<B1-Motion>", self.on_drag)  # move with button 1 held
        self.canvas.bind("<Motion>", self.report)

    def on_press(self, event):
        self.last = (event.x, event.y)

    def on_drag(self, event):
        x0, y0 = self.last
        # Each segment becomes a canvas item; Tk remembers and redraws it.
        self.canvas.create_line(x0, y0, event.x, event.y, fill="blue",
                                width=12, capstyle=tk.ROUND)
        self.last = (event.x, event.y)
        self.report(event)

    def report(self, event):
        self.xy.config(text=f"X: {event.x}, Y: {event.y}")


def self_test():
    root = tk.Tk()
    root.withdraw()
    d = Doodle(root)
    d.pack()

    def at(x, y):  # a stand-in for the event object Tk would pass
        return SimpleNamespace(x=x, y=y)
```

Listing 10.16 (Continued) *doodle.py: drawing with the mouse on a Tkinter canvas; every segment is a line item that Tk keeps.*

```
    d.on_press(at(50, 100))
    d.on_drag(at(100, 100))
    d.on_drag(at(150, 150))
    d.report(at(200, 60))
    items = d.canvas.find_all()
    print("canvas items:", len(items))
    print("label:       ", d.xy.cget("text"))
    print("item 1:      ", d.canvas.type(items[0]), d.canvas.coords(items[0]),
          d.canvas.itemcget(items[0], "fill"))
    root.destroy()


def main():
    if "--selftest" in sys.argv:
        self_test()
        return
    root = tk.Tk()
    root.title("Mouse motion event")
    Doodle(root).pack()
    root.mainloop()


if __name__ == "__main__":
    main()
```

```
canvas items: 2
label:        X: 200, Y: 60
item 1:       line [50.0, 100.0, 100.0, 100.0] blue
```

The event sequence `"<B1-Motion>"` means “motion with button 1 held down”, Tk’s name for a drag. The self-test builds stand-in event objects with `types.SimpleNamespace(x=..., y=...)`, which have exactly the attributes the callbacks use, and calls the callbacks in the order Tk would: one press and two drags produce two line items, the first from (50,100) to (100,100) in blue.

10.11 The Event Dispatch Thread and the Thread Rule

We now return to the question left open in Section 10.4: why does `main` create its window with `SwingUtilities.invokeLater`? The answer is the most important rule of GUI programming, and it holds in Swing, in Tk, in the browser, on Android and on iOS alike.

10.11.1 One thread owns the window

A **thread** is a sequence of instructions executing inside a program; a program can have several threads running at the same time, sharing its objects. Chapter 14 is devoted to threads and the troubles they bring. Here we need only one fact about them: when two threads change the same objects at the same time, the objects can be left in an inconsistent state, unless every access is carefully coordinated.

GUI toolkits contain enormous numbers of objects that change constantly — every component’s text, size, position, color, focus, selection. Coordinating every access among several threads would make every toolkit slow and complicated, and experience from the 1990s showed that multithreaded toolkits deadlock easily [15]. So virtually every toolkit makes the same decision: **all access to the GUI happens on one thread**. In Swing that thread is the **event dispatch thread** (EDT), the thread that runs the event loop of **Pseudocode 10.1**. In Tkinter it is the thread that runs `mainloop()`, normally the program’s main thread.

> **DEFINITION. The thread rule.** Swing components, and Tkinter widgets, may be created, read and changed only on the GUI thread — the event dispatch thread in Swing, the thread running `mainloop()` in Tkinter. Conversely, the GUI thread must never be kept busy for long, because while it is busy no events are handled and nothing is repainted.

The two halves of the rule pull in opposite directions, and the art of GUI programming is to satisfy both.

- **Everything GUI on the GUI thread.** Listeners already run on the EDT (the event loop calls them), so code inside listeners may touch components freely. Code that runs *elsewhere* — `main`, which runs on the main thread, or a background thread — must send GUI work to the EDT. `SwingUtilities.invokeLater(r)` puts the `Runnable` `r` into the event queue and returns at once; `SwingUtilities.invokeAndWait(r)` does the same and waits until `r` has run. That is why `main` builds the window with `invokeLater` [16].
- **Nothing slow on the GUI thread.** A listener that reads a large file, queries a database, calls a web service or computes for a second holds the event loop on line 7 of **Pseudocode 10.1** for the whole time. The window stops repainting (it goes gray or white if uncovered), buttons do not respond, and the operating system may offer to kill the "not responding" program. As a rule of thumb, a reaction within about 0.1 seconds feels instantaneous to a user, and after about 1 second the user's flow of thought is interrupted [17]; a handler should finish well within the first limit.

10.11.2 Slow work, the wrong way and the right way

Listing 10.17 has two buttons that compute the same thing: the laundry's revenue for a month, where reading each day's orders takes 20 milliseconds (simulated with `Thread.sleep`), so the whole tally takes about 600 ms. With daily revenue $100{,}000 + 5{,}000\,(d \bmod 7)$ for day d, the month total is

$$\sum_{d=1}^{30} (100{,}000 + 5{,}000\,(d \bmod 7)) = 30 \times 100{,}000 + 5{,}000 \times 87 = 3{,}435{,}000 \tag{10.4}$$

because the remainders $d \bmod 7$ for $d = 1, \ldots, 30$ are four full weeks of $1 + 2 + \cdots + 6 + 0 = 21$, plus $1 + 2$ for days 29 and 30, which is $4 \times 21 + 3 = 87$. Equation (10.4) is what both buttons must display. *Tally (freezes)* computes it right inside its listener. *Tally the month* hands it to a `SwingWorker`, Swing's helper for running a task on a background thread and delivering the result back on the EDT.

Listing 10.17 `SlowWork.java`: slow work on the EDT freezes the window; a `SwingWorker` runs it on a worker thread and shows the result on the EDT.

```
import java.util.Locale;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutionException;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
import javax.swing.SwingWorker;

public class SlowWork extends JPanel {
    private final JButton start = new JButton("Tally the month");
    private final JButton freeze = new JButton("Tally (freezes)");
    private final JLabel status = new JLabel("Ready");
    private Runnable onDone = () -> { };  // lets the self-test know we finished

    public SlowWork() {
        add(start);
        add(freeze);
        add(status);
        freeze.addActionListener(e -> {  // WRONG: slow work on the EDT
            log("click handled on");
            long total = tally();         // the window is frozen for the whole tally
            status.setText("Month total: " + rupiah(total));
        });
        start.addActionListener(e -> {   // RIGHT: hand the slow work to a worker
            log("click handled on");
            start.setEnabled(false);
            status.setText("Working...");
            new SwingWorker<Long, Void>() {
                @Override
                protected Long doInBackground() {  // runs on a worker thread
                    return tally();
                }

                @Override
                protected void done() {            // runs on the EDT again
                    log("done() runs on");
                    try {
                        status.setText("Month total: " + rupiah(get()));
                    } catch (InterruptedException | ExecutionException ex) {
                        status.setText("Failed: " + ex.getMessage());
                    }
                    start.setEnabled(true);
                    onDone.run();
                }
```

Listing 10.17 (Continued) *SlowWork.java: slow work on the EDT freezes the window; a SwingWorker runs it on a worker thread and shows the result on the EDT.*

```
            }.execute();
        });
    }

    static void log(String what) {
        System.out.println(what + " " + Thread.currentThread().getName());
    }

    static String rupiah(long amount) {
        return String.format(Locale.US, "Rp%,d", amount);
    }

    static long tally() {  // pretend each day's orders take 20 ms to read
        log("tally runs on");
        long total = 0;
        for (int day = 1; day <= 30; day++) {
            try {
                Thread.sleep(20);
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
                return total;
            }
            total += 100_000 + 5_000 * (day % 7);
        }
        return total;
    }

    public static void main(String[] args) throws Exception {
        if (args.length > 0 && args[0].equals("--selftest")) {
            System.setProperty("java.awt.headless", "true");
            selfTest();
        } else {
            SwingUtilities.invokeLater(() -> {
                JFrame.setDefaultLookAndFeelDecorated(true);
                JFrame frame = new JFrame("Slow work");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.add(new SlowWork());
                frame.pack();
                frame.setVisible(true);
            });
        }
    }
```

Listing 10.17 (Continued) *SlowWork.java: slow work on the EDT freezes the window; a SwingWorker runs it on a worker thread and shows the result on the EDT.*

```
    static void selfTest() throws Exception {
        CountDownLatch finished = new CountDownLatch(1);
        SlowWork[] holder = new SlowWork[1];
        SwingUtilities.invokeAndWait(() -> {
            SlowWork w = new SlowWork();
            holder[0] = w;
            w.onDone = finished::countDown;
            long t0 = System.nanoTime();
            w.freeze.doClick();
            long ms = (System.nanoTime() - t0) / 1_000_000;
            System.out.println("EDT blocked for at least 600 ms: " + (ms >= 600));
            System.out.println("status: " + w.status.getText());
            t0 = System.nanoTime();
            w.start.doClick();
            ms = (System.nanoTime() - t0) / 1_000_000;
            System.out.println("EDT free again within 100 ms: " + (ms < 100));
            System.out.println("status: " + w.status.getText());
        });
        finished.await();  // the main thread waits; the EDT stays free
        SwingUtilities.invokeAndWait(() ->
                System.out.println("status: " + holder[0].status.getText()));
    }
}
```

```
click handled on AWT-EventQueue-0
tally runs on AWT-EventQueue-0
EDT blocked for at least 600 ms: true
status: Month total: Rp3,435,000
click handled on AWT-EventQueue-0
EDT free again within 100 ms: true
status: Working...
tally runs on SwingWorker-pool-1-thread-1
done() runs on AWT-EventQueue-0
status: Month total: Rp3,435,000
```

The log names the thread of each step. For the frozen button, the click handler *and* the tally ran on `AWT-EventQueue-0`, the EDT, which was blocked for at least 600 ms — a real window would have been dead for that time. For the right button, the handler disabled the button, showed *Working…*, started the worker and returned within 100 ms; the tally ran on `SwingWorker-pool-1-thread-1`; and `done()` ran back on the EDT, where it may safely set the label. `get()` returns the result of `doInBackground()`, or throws an `ExecutionException` wrapping its exception. (Lines from different threads may interleave differently on your machine; the thread names will not change.)

A `SwingWorker<T, V>` is generic, in the sense of Chapter 9: `T` is the type of the overall result (here `Long`) and `V` the type of intermediate results, which a worker can send with `publish(...)` and receive on the EDT in `process(...)` to drive a progress bar; we do not need them (`Void`). **Figure 10.17** shows both runs on a time line.

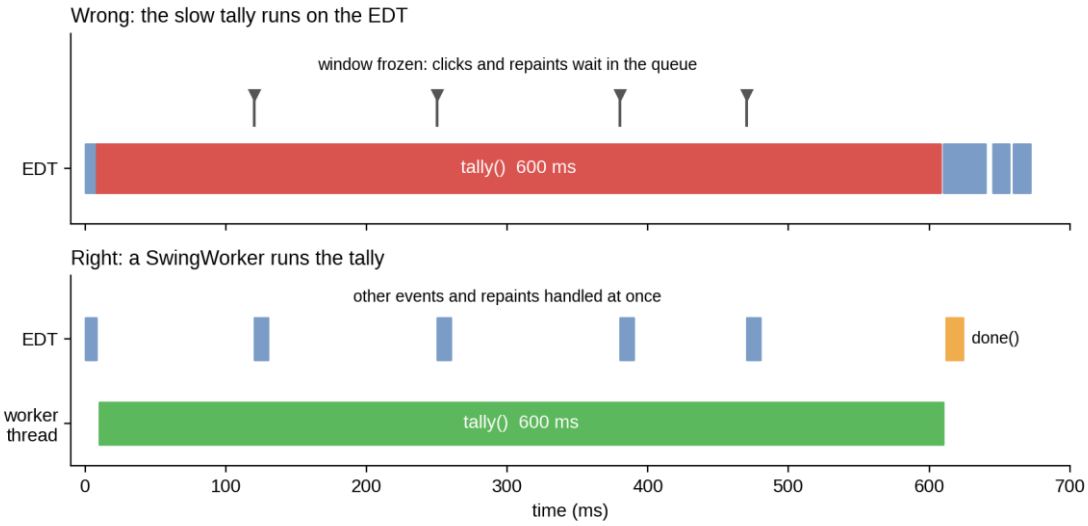


Figure 10.17 The thread rule on a time line. Top: the tally runs on the EDT, which cannot handle clicks or repaints for 600 ms. Bottom: a worker thread runs the tally while the EDT keeps handling events; `done()` returns to the EDT to show the result.

Tkinter has no `SwingWorker`, but the recipe is the same and is worth knowing by heart: *start a thread for the work; let the thread put its result into a thread-safe queue; let the GUI thread poll the queue with a timer*. **Listing 10.18** follows the recipe.

Listing 10.18 `slow_work.py`: a worker thread computes, a `queue.Queue` carries the result, and `after` polls the queue on the GUI thread.

```
"""slow_work.py: keep a Tkinter window responsive while slow work runs."""
import queue
import sys
import threading
import time
import tkinter as tk


def log(what):
    print(what, threading.current_thread().name)


def tally():  # pretend each day's orders take 20 ms to read
    log("tally runs on")
    total = 0
    for day in range(1, 31):
        time.sleep(0.02)
        total += 100_000 + 5_000 * (day % 7)
    return total


class SlowWork(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.start = tk.Button(self, text="Tally the month", command=self.on_start)
        self.freeze = tk.Button(self, text="Tally (freezes)", command=self.on_freeze)
        self.status = tk.Label(self, text="Ready")
        for w in (self.start, self.freeze, self.status):
            w.pack(side=tk.LEFT, padx=4, pady=4)
        self.results = queue.Queue()  # the only thing both threads touch

    def on_freeze(self):  # WRONG: slow work inside the event loop
        log("click handled on")
        self.status.config(text=f"Month total: Rp{tally():,}")

    def on_start(self):  # RIGHT: slow work on a worker thread
        log("click handled on")
        self.start.config(state=tk.DISABLED)
        self.status.config(text="Working...")
        threading.Thread(target=self.work, daemon=True).start()
        self.after(50, self.poll)

    def work(self):  # worker thread: no widget calls here!
        self.results.put(tally())
```

Listing 10.18 (Continued) *slow_work.py: a worker thread computes, a queue.Queue carries the result, and after polls the queue on the GUI thread.*

```
    def poll(self):  # event-loop thread: check the queue every 50 ms
        try:
            total = self.results.get_nowait()
        except queue.Empty:
            self.after(50, self.poll)
            return
        log("result shown on")
        self.status.config(text=f"Month total: Rp{total:,}")
        self.start.config(state=tk.NORMAL)


def self_test():
    root = tk.Tk()
    root.withdraw()
    w = SlowWork(root)
    w.pack()
    t0 = time.perf_counter()
    w.freeze.invoke()
    print("loop blocked for at least 600 ms:", time.perf_counter() - t0 >= 0.6)
    print("status:", w.status.cget("text"))
    t0 = time.perf_counter()
    w.start.invoke()
    print("loop free again within 100 ms:", time.perf_counter() - t0 < 0.1)
    print("status:", w.status.cget("text"))

    def finish():
        if w.status.cget("text").startswith("Month"):
            print("status:", w.status.cget("text"))
            root.destroy()  # ends mainloop
        else:
            root.after(50, finish)

    root.after(50, finish)
    root.mainloop()


def main():
    if "--selftest" in sys.argv:
        self_test()
        return
    root = tk.Tk()
    root.title("Slow work")
    SlowWork(root).pack()
    root.mainloop()
```

Listing 10.18 (Continued) *slow_work.py: a worker thread computes, a queue.Queue carries the result, and after polls the queue on the GUI thread.*

```
if __name__ == "__main__":
    main()
```

```
click handled on MainThread
tally runs on MainThread
loop blocked for at least 600 ms: True
status: Month total: Rp3,435,000
click handled on MainThread
loop free again within 100 ms: True
status: Working...
tally runs on Thread-1 (work)
result shown on MainThread
status: Month total: Rp3,435,000
```

`threading.Thread(target=self.work, daemon=True).start()` runs `work` on a new thread (a *daemon* thread does not keep the program alive after the window closes). `work` touches no widget; it puts the total into `self.results`, a thread-safe `queue.Queue`. On the GUI thread, `self.after(50, self.poll)` asks Tk's loop to call `poll` in 50 ms; `poll` checks the queue without waiting and either reschedules itself or shows the result. The log confirms that the widgets were updated on `MainThread`, while the tally ran on `Thread-1 (work)`.

Table 10.5 collects the tools of the rule for both toolkits.

Table 10.5 The thread rule's tools in Swing and in Tkinter.

Task	Swing	Tkinter
Run GUI code from another thread, later	`SwingUtilities.invokeLater(r)`	put a message in a `queue.Queue`; poll it with `after`
Run GUI code from another thread and wait	`SwingUtilities.invokeAndWait(r)`	(avoid; use the queue)
Am I on the GUI thread?	`SwingUtilities.isEventDispatchThread()`	`threading.current_thread() is threading.main_thread()`
Slow work in the background	`SwingWorker.doInBackground()`, result in `done()`	`threading.Thread(target=...)` plus a queue
Progress from the background	`publish(...)` / `process(...)`	put progress messages in the same queue
A timer on the GUI thread	`javax.swing.Timer(ms, listener)`	`widget.after(ms, callback)`

10.12 Profitina Laundry: The Order-Entry Window

We can now build the window that the laundry counter actually needs. The requirements come from the staff:

- **The form.** Customer name, phone number, service (Wash & Fold, Iron Only, or Wash, Dry & Iron), weight in kilograms, and an *Express* box. The prices are those of Chapter 7: Wash & Fold Rp7,000 per kilogram with a 3 kg minimum, ready in 48 hours; Iron Only Rp5,000 per kilogram with a 2 kg minimum, ready in 24 hours; Wash, Dry & Iron Rp10,000 per kilogram with a 3 kg minimum, ready in 72 hours. Express costs 50% more and is four times faster, but never faster than 6 hours.
- **A live price.** As soon as the weight, the service or the express box changes, the window shows the price, the chargeable weight and the turnaround — before anything is added — so the staff can tell the customer.
- **Validation.** *Add order* accepts the order only if the name is present (at most 40 characters), the phone number has 10 to 13 digits starting with 0 (spaces and dashes are allowed and removed), and the weight is a number above 0 and at most 50 kg. Indonesian staff often type a decimal comma, so `4,2` must be accepted as well as `4.2`. On a mistake, the window says what is wrong in red and puts the cursor in the field to fix; nothing else changes.
- **The list.** Each accepted order gets the next number (ORD-0001, ORD-0002, …) and a row in a list: number, customer, service code (prefixed `EX-` for express), chargeable weight, price and turnaround. The form is then cleared for the next customer.
- **Remove and total.** A *Remove selected* button, enabled only while a row is selected, removes that order. The footer always shows the number of orders and their total.

10.12.1 The window, described precisely

This is the window as the staff see it, with three orders entered, a fourth being typed, and the second row selected:

```
+-- Profitina Laundry - Order Entry -------------------------------[_][O][X]+
|  Customer     [Dewi Lestari        ]                                      |
|  Phone        [0813 5555 0101      ]                                      |
|  Service      [Wash & Fold (Rp7,000/kg)       v]                          |
|  Weight (kg)  [6.3    ]                   [x] Express (+50%, 4x faster)    |
|               Price: Rp68,250 for 6.5 kg, ready in 12 h                    |
|               ( Add order ) ( Clear )                                      |
|  Added ORD-0003 for Ayu Lestari.                    (in green)             |
| +- Orders ---------------------------------------------------------------+ |
| | ORD-0001  Siti Aminah    WF       4.5 kg    Rp31,500  48 h             | |
| |#ORD-0002  Budi Santoso   EX-WDI   3.0 kg    Rp45,000  18 h#############| |
| | ORD-0003  Ayu Lestari    IR       2.0 kg    Rp10,000  24 h             | |
| |                                                                        | |
| +------------------------------------------------------------------------+ |
|  ( Remove selected )                          3 orders   Total: Rp86,500   |
+----------------------------------------------------------------------------+
```

From top to bottom: four labeled fields in two columns, with *Express* in a third column beside the weight; the live price; the two buttons; the status line, green after a success and red after an error. In the middle, a list titled *Orders* in a fixed-width font, with a scroll bar when needed; it grows when the window is enlarged, and the selected row, drawn with `#`, is highlighted. At the bottom, *Remove selected* on the left and the count and total on the right. **Figure 10.18** is the real Swing window in exactly this state, and **Figure 10.19** the Tkinter one.

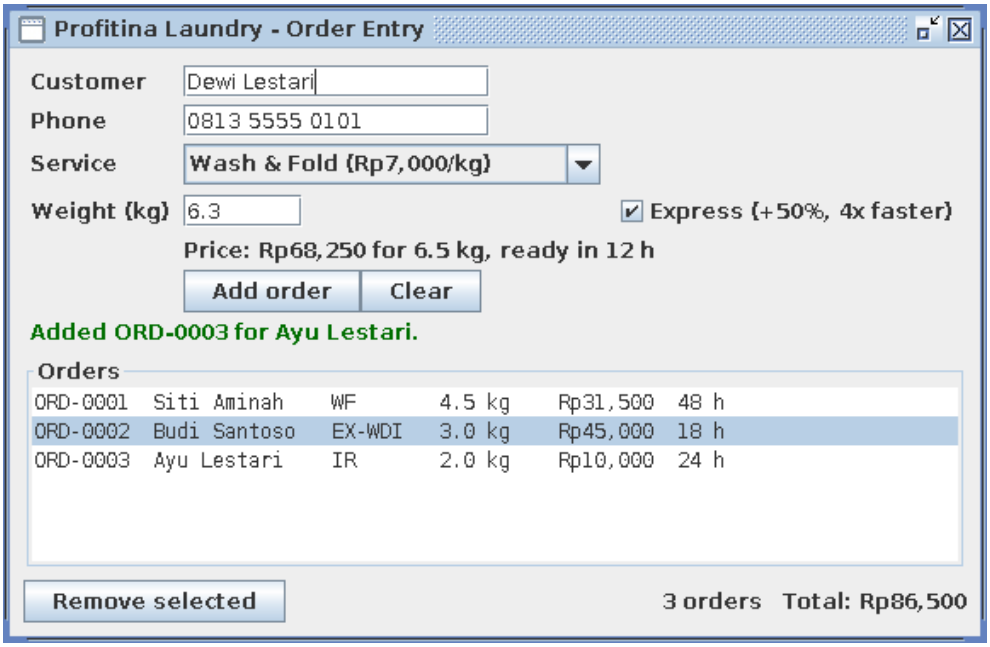


Figure 10.18 The Swing order-entry window with three orders, a fourth being typed (the live price already shows Rp68,250), and the second row selected.

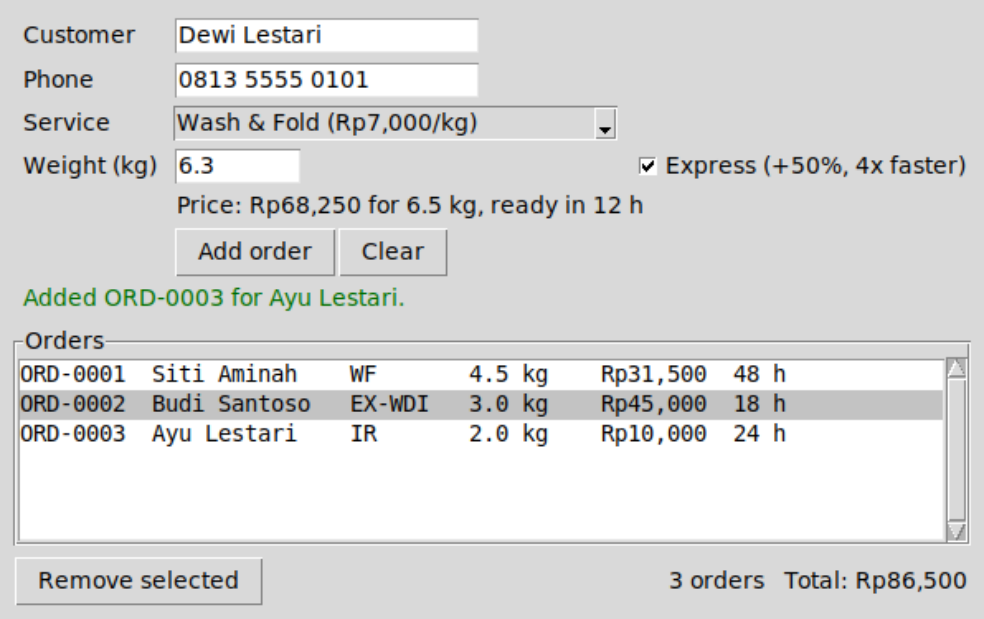


Figure 10.19 The Tkinter order-entry window in the same state. (The screenshot shows the window's contents without the title bar that the operating system adds.)

10.12.2 The prices, precisely

The pricing rules are Chapter 7's, restated for one order line. For a service with rate r rupiah per kilogram and minimum $w_{\min}$, a bag weighing w kilograms is charged for the **chargeable weight** $\hat{w}$: the weight raised to the minimum and rounded up to the next half kilogram,

$$\hat{w} = \frac{\lceil 2\max(w, w_{\min}) \rceil}{2} \tag{10.5}$$

and the normal price and turnaround are

$$P = r \cdot \hat{w}, \qquad H = h \tag{10.6}$$

where h is the service's normal turnaround in hours. For an express order, price and turnaround become

$$P_{\text{ex}} = \lfloor 3P/2 \rfloor, \qquad H_{\text{ex}} = \max(6, \lfloor h/4 \rfloor) \tag{10.7}$$

Let us check the three orders of **Figure 10.18**. Siti's 4.2 kg of Wash & Fold: by Equation (10.5), $\max(4.2,3) = 4.2$, $2 \times 4.2 = 8.4$, rounded up to 9, halved to $\hat{w} = 4.5$ kg; by Equation (10.6), $P = 7000 \times 4.5 = 31{,}500$, ready in 48 h. Budi's 2.6 kg of Wash, Dry & Iron, express: $\max(2.6,3) = 3$, so $\hat{w} = 3.0$ and $P = 10{,}000 \times 3 = 30{,}000$; by Equation (10.7), $P_{\text{ex}} = \lfloor 90{,}000/2 \rfloor = 45{,}000$ and $H_{\text{ex}} = \max(6, 72/4) = 18$ h. Ayu's 1.3 kg of ironing is raised to the 2 kg minimum: $5000 \times 2 = 10{,}000$, ready in 24 h. The total is $31{,}500 + 45{,}000 + 10{,}000 = 86{,}500$. And Dewi's order in progress, 6.3 kg of Wash & Fold, express: $\lceil 12.6 \rceil = 13$, $\hat{w} = 6.5$, $P = 45{,}500$, $P_{\text{ex}} = 68{,}250$, ready in $\max(6,12) = 12$ h — exactly what the live price shows.

In code, $\lceil 2\max(w, w_{\min}) \rceil$ is computed as `Math.ceil(2 * Math.max(kg, s.minKg) - 1e-9)`. The tiny 10^{-9} protects against binary rounding: a weight typed as `3.5` is stored exactly, but a weight such as `4.15` is stored as a binary fraction a hair above or below 4.15, and doubling a value a hair *above* a half-kilogram boundary would otherwise round up one step too far. Chapter 3 explained

why decimal fractions are rarely exact in binary. Because every rate is a multiple of 2, $r \cdot \widehat{w} = r \cdot \lceil \dots \rceil / 2$ is computed exactly in integer arithmetic as `s.rate * halfKg / 2`.

10.12.3 The design: a model that knows nothing about windows

The single most important design decision is visible in **Figure 10.20**: the program is split into a **model**, which knows about services, prices, validation and the list of orders but imports nothing from Swing or Tkinter, and a **view**, the window class, which knows about components and events and asks the model for every decision.

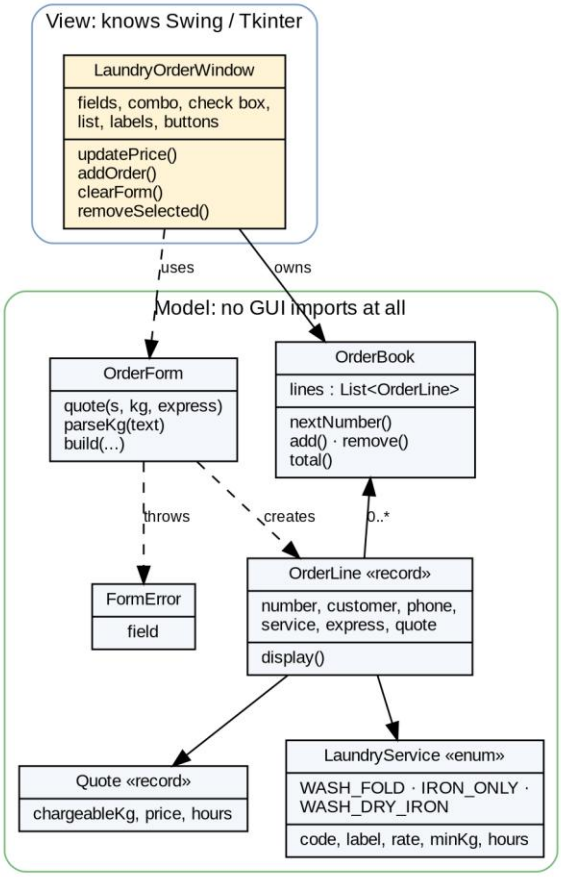


Figure 10.20 The design of the order-entry program. The view owns an `OrderBook` and uses `OrderForm` for every calculation and check; the model classes never mention a window.

- **`LaundryService`** is an enum whose three constants carry their code, label, rate, minimum and turnaround. Its `toString()` produces the text the combo box shows, such as *Wash & Fold (Rp7,000/kg)*.
- **`Quote`** is a small immutable value: chargeable weight, price and turnaround.
- **`OrderLine`** is an immutable record of one accepted order, with `code()` and `display()`, which formats the row shown in the list.
- **`FormError`** is an exception that carries a user-friendly message *and* the name of the field to fix, so the view knows where to put the cursor.
- **`OrderForm`** holds the rules: `quote` implements Equation (10.5), Equation (10.6) and Equation (10.7); `parseKg` turns the text of the weight field into a number or throws `FormError`; `build` validates the whole form and returns an `OrderLine`.
- **`OrderBook`** holds the accepted orders, hands out order numbers and computes the total.

- `LaundryOrderWindow` builds the components, connects each event to a short handler, and turns the model's answers into text and colors.

Why bother? Because each half can now be understood, changed and tested alone. The pricing rules can be tested without creating a single component; the window can be redesigned — a different layout, a different toolkit — without touching a pricing rule. Chapter 11 makes this separation into a pattern, Model–View–Controller [18], and shows how the model can notify the view of changes by itself.

10.12.4 The Swing program

Listing 10.19 is the whole program in one file, `LaundryOrderWindow.java`. The model classes are package-private top-level classes (Chapter 7) in the same file; in a larger program each would have its own file.

Listing 10.19 `LaundryOrderWindow.java`: the Profitina Laundry order-entry window in Swing, with its model classes and a `--selftest` mode.

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Locale;
import javax.swing.BorderFactory;
import javax.swing.DefaultListModel;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
import javax.swing.ListSelectionModel;
import javax.swing.SwingUtilities;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;

// ---------- The model: plain Java, no Swing ----------

enum LaundryService {
    WASH_FOLD("WF", "Wash & Fold", 7000, 3.0, 48),
    IRON_ONLY("IR", "Iron Only", 5000, 2.0, 24),
    WASH_DRY_IRON("WDI", "Wash, Dry & Iron", 10000, 3.0, 72);

    final String code;
    final String label;
    final long rate;     // rupiah per kilogram
    final double minKg;  // minimum chargeable weight
    final int hours;     // normal turnaround
```

Listing 10.19 (Continued) *LaundryOrderWindow.java: the Profitina Laundry order-entry window in Swing, with its model classes and a --selftest mode.*

```
    LaundryService(String code, String label, long rate, double minKg, int hours) {
        this.code = code;
        this.label = label;
        this.rate = rate;
        this.minKg = minKg;
        this.hours = hours;
    }

    @Override
    public String toString() {  // what the combo box shows
        return label + " (" + LaundryOrderWindow.rupiah(rate) + "/kg)";
    }
}

record Quote(double chargeableKg, long price, int hours) { }

record OrderLine(String number, String customer, String phone,
                 LaundryService service, boolean express, Quote quote) {
    String code() {
        return (express ? "EX-" : "") + service.code;
    }

    String display() {
        return String.format(Locale.US, "%s  %-14.14s %-6s %5.1f kg %11s %3d h",
                number, customer, code(), quote.chargeableKg(),
                LaundryOrderWindow.rupiah(quote.price()), quote.hours());
    }
}

class FormError extends Exception {  // a message for the user, and the field to fix
    final String field;

    FormError(String field, String message) {
        super(message);
        this.field = field;
    }
}

final class OrderForm {
    private OrderForm() { }
```

Listing 10.19 (Continued) *LaundryOrderWindow.java: the Profitina Laundry order-entry window in Swing, with its model classes and a --selftest mode.*

```
    static Quote quote(LaundryService s, double kg, boolean express) {
        long halfKg = (long) Math.ceil(2 * Math.max(kg, s.minKg) - 1e-9);
        long price = s.rate * halfKg / 2;
        int hours = s.hours;
        if (express) {
            price = 3 * price / 2;
            hours = Math.max(6, hours / 4);
        }
        return new Quote(halfKg / 2.0, price, hours);
    }

    static double parseKg(String text) throws FormError {
        String t = text.strip().replace(',', '.');  // accept 4,2 as well as 4.2
        if (t.isEmpty()) {
            throw new FormError("weight", "Weight is required.");
        }
        if (!t.matches("\\d+(\\.\\d+)?")) {
            throw new FormError("weight", "Weight must be a number, e.g. 4.5");
        }
        double kg = Double.parseDouble(t);
        if (kg <= 0) {
            throw new FormError("weight", "Weight must be more than 0 kg.");
        }
        if (kg > 50) {
            throw new FormError("weight", "Weight must be at most 50 kg.");
        }
        return kg;
    }

    static OrderLine build(String number, String name, String phone,
                           LaundryService s, String kgText, boolean express)
            throws FormError {
        String n = name.strip();
        if (n.isEmpty()) {
            throw new FormError("name", "Customer name is required.");
        }
        if (n.length() > 40) {
            throw new FormError("name", "Customer name must be at most 40 characters.");
        }
        String p = phone.replaceAll("[\\s-]", "");
        if (!p.matches("0\\d{9,12}")) {
            throw new FormError("phone", "Phone must be 10 to 13 digits starting with 0.");
        }
        double kg = parseKg(kgText);
        return new OrderLine(number, n, p, s, express, quote(s, kg, express));
    }
```

Listing 10.19 (Continued) *LaundryOrderWindow.java: the Profitina Laundry order-entry window in Swing, with its model classes and a --selftest mode.*

```
}

class OrderBook {
    private final List<OrderLine> lines = new ArrayList<>();
    private int next = 1;

    String nextNumber() {
        return String.format("ORD-%04d", next);
    }

    void add(OrderLine line) {
        lines.add(line);
        next++;
    }

    OrderLine remove(int index) {
        return lines.remove(index);
    }

    List<OrderLine> lines() {
        return Collections.unmodifiableList(lines);
    }

    long total() {
        long t = 0;
        for (OrderLine l : lines) {
            t += l.quote().price();
        }
        return t;
    }
}

// ---------- The view: Swing components and their listeners ----------

public class LaundryOrderWindow extends JPanel {
    private static final Color ERROR = new Color(170, 0, 0);
    private static final Color OK = new Color(0, 110, 0);
```

Listing 10.19 (Continued) *LaundryOrderWindow.java: the Profitina Laundry order-entry window in Swing, with its model classes and a --selftest mode.*

```
    private final OrderBook book = new OrderBook();
    final JTextField nameField = new JTextField(16);
    final JTextField phoneField = new JTextField(16);
    final JComboBox<LaundryService> serviceBox = new JComboBox<>(LaundryService.values());
    final JTextField weightField = new JTextField(6);
    final JCheckBox expressBox = new JCheckBox("Express (+50%, 4x faster)");
    final JLabel priceLabel = new JLabel("Price: -");
    final JButton addButton = new JButton("Add order");
    final JButton clearButton = new JButton("Clear");
    final JLabel statusLabel = new JLabel(" ");
    final DefaultListModel<String> listModel = new DefaultListModel<>();
    final JList<String> orderList = new JList<>(listModel);
    final JButton removeButton = new JButton("Remove selected");
    final JLabel totalLabel = new JLabel("0 orders   Total: Rp0");

    public LaundryOrderWindow() {
        super(new BorderLayout(6, 6));
        setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8));
        add(buildForm(), BorderLayout.PAGE_START);
        orderList.setFont(new Font(Font.MONOSPACED, Font.PLAIN, 12));
        orderList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        orderList.setVisibleRowCount(6);
        JScrollPane scroller = new JScrollPane(orderList);
        scroller.setBorder(BorderFactory.createTitledBorder("Orders"));
        add(scroller, BorderLayout.CENTER);
        JPanel footer = new JPanel(new BorderLayout());
        footer.add(removeButton, BorderLayout.LINE_START);
        footer.add(totalLabel, BorderLayout.LINE_END);
        add(footer, BorderLayout.PAGE_END);
        removeButton.setEnabled(false);
        wireEvents();
    }

    private JPanel buildForm() {
        JPanel form = new JPanel(new GridBagLayout());
        GridBagConstraints c = new GridBagConstraints();
        c.insets = new Insets(2, 4, 2, 4);
        c.anchor = GridBagConstraints.LINE_START;
        String[] labels = {"Customer", "Phone", "Service", "Weight (kg)"};
        java.awt.Component[] fields = {nameField, phoneField, serviceBox, weightField};
        for (int row = 0; row < labels.length; row++) {
            c.gridx = 0;
            c.gridy = row;
            form.add(new JLabel(labels[row]), c);
            c.gridx = 1;
            form.add(fields[row], c);
```

Listing 10.19 (Continued) *LaundryOrderWindow.java: the Profitina Laundry order-entry window in Swing, with its model classes and a --selftest mode.*

```
        }
        c.gridx = 2;
        c.gridy = 3;
        form.add(expressBox, c);
        c.gridx = 1;
        c.gridy = 4;
        c.gridwidth = 2;
        form.add(priceLabel, c);
        JPanel buttons = new JPanel(new FlowLayout(FlowLayout.LEFT, 0, 0));
        buttons.add(addButton);
        buttons.add(clearButton);
        c.gridy = 5;
        form.add(buttons, c);
        c.gridx = 0;
        c.gridy = 6;
        c.gridwidth = 3;
        form.add(statusLabel, c);
        return form;
    }

    private void wireEvents() {
        weightField.getDocument().addDocumentListener(new DocumentListener() {
            @Override public void insertUpdate(DocumentEvent e) { updatePrice(); }
            @Override public void removeUpdate(DocumentEvent e) { updatePrice(); }
            @Override public void changedUpdate(DocumentEvent e) { updatePrice(); }
        });
        serviceBox.addActionListener(e -> updatePrice());
        expressBox.addActionListener(e -> updatePrice());
        addButton.addActionListener(e -> addOrder());
        weightField.addActionListener(e -> addOrder());  // Enter in the weight field
        clearButton.addActionListener(e -> clearForm());
        orderList.addListSelectionListener(
                e -> removeButton.setEnabled(orderList.getSelectedIndex() >= 0));
        removeButton.addActionListener(e -> removeSelected());
    }

    private LaundryService service() {
        return (LaundryService) serviceBox.getSelectedItem();
    }
```

Listing 10.19 (Continued) *LaundryOrderWindow.java: the Profitina Laundry order-entry window in Swing, with its model classes and a --selftest mode.*

```
    void updatePrice() {
        try {
            double kg = OrderForm.parseKg(weightField.getText());
            Quote q = OrderForm.quote(service(), kg, expressBox.isSelected());
            priceLabel.setText(String.format(Locale.US, "Price: %s for %.1f kg, ready in %d
h",
                    rupiah(q.price()), q.chargeableKg(), q.hours()));
        } catch (FormError e) {
            priceLabel.setText("Price: -");
        }
    }

    void addOrder() {
        try {
            OrderLine line = OrderForm.build(book.nextNumber(), nameField.getText(),
                    phoneField.getText(), service(), weightField.getText(),
                    expressBox.isSelected());
            book.add(line);
            listModel.addElement(line.display());
            clearForm();
            say(OK, "Added " + line.number() + " for " + line.customer() + ".");
            nameField.requestFocusInWindow();
        } catch (FormError e) {
            say(ERROR, e.getMessage());
            switch (e.field) {
                case "name" -> nameField.requestFocusInWindow();
                case "phone" -> phoneField.requestFocusInWindow();
                default -> weightField.requestFocusInWindow();
            }
        }
        refreshTotal();
    }

    void clearForm() {
        nameField.setText("");
        phoneField.setText("");
        serviceBox.setSelectedIndex(0);
        weightField.setText("");
        expressBox.setSelected(false);
        priceLabel.setText("Price: -");
        say(OK, " ");
    }
```

Listing 10.19 (Continued) *LaundryOrderWindow.java: the Profitina Laundry order-entry window in Swing, with its model classes and a --selftest mode.*

```
    void removeSelected() {
        int i = orderList.getSelectedIndex();
        if (i < 0) {
            return;
        }
        OrderLine gone = book.remove(i);
        listModel.remove(i);
        say(OK, "Removed " + gone.number() + ".");
        refreshTotal();
    }

    private void say(Color color, String message) {
        statusLabel.setForeground(color);
        statusLabel.setText(message);
    }

    private void refreshTotal() {
        int n = book.lines().size();
        totalLabel.setText(n + (n == 1 ? " order" : " orders")
                + "    Total: " + rupiah(book.total()));
    }

    static String rupiah(long amount) {
        return String.format(Locale.US, "Rp%,d", amount);
    }

    public static void main(String[] args) throws Exception {
        if (args.length > 0 && args[0].equals("--selftest")) {
            System.setProperty("java.awt.headless", "true");
            SwingUtilities.invokeAndWait(LaundryOrderWindow::selfTest);
        } else {
            SwingUtilities.invokeLater(() -> {
                JFrame.setDefaultLookAndFeelDecorated(true);
                JFrame frame = new JFrame("Profitina Laundry - Order Entry");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setContentPane(new LaundryOrderWindow());
                frame.pack();
                frame.setLocationRelativeTo(null);  // center on the screen
                frame.setVisible(true);
            });
        }
    }

    // ---------- Self-test: drive the real components without a screen ----------
```

Listing 10.19 (Continued) *LaundryOrderWindow.java: the Profitina Laundry order-entry window in Swing, with its model classes and a --selftest mode.*

```
    void fill(String name, String phone, int service, String kg, boolean express) {
        nameField.setText(name);
        phoneField.setText(phone);
        serviceBox.setSelectedIndex(service);
        weightField.setText(kg);
        if (expressBox.isSelected() != express) {
            expressBox.doClick();
        }
    }

    void report(String step) {
        System.out.println("== " + step);
        System.out.println("   price:  " + priceLabel.getText());
        System.out.println("   status: " + statusLabel.getText());
    }

    static void selfTest() {
        LaundryOrderWindow w = new LaundryOrderWindow();
        w.addButton.doClick();
        w.report("1. Add with an empty form");
        w.fill("Siti Aminah", "0812-3456-7890", 0, "4,2", false);
        w.report("2. Siti, Wash & Fold, 4,2 kg (typed, not yet added)");
        w.addButton.doClick();
        w.report("   ...Add order clicked");
        w.fill("Budi Santoso", "081298765432", 2, "2.6", true);
        w.report("3. Budi, Wash, Dry & Iron, 2.6 kg, express (typed)");
        w.addButton.doClick();
        w.report("   ...Add order clicked");
        w.fill("Ayu Lestari", "12345", 1, "1.3", false);
        w.addButton.doClick();
        w.report("4. Ayu with phone 12345");
        for (String kg : new String[] {"abc", "0", "75"}) {
            w.fill("Ayu Lestari", "0857 1111 2222", 1, kg, false);
            w.addButton.doClick();
            w.report("5. Ayu with weight " + kg);
        }
        w.fill("Ayu Lestari", "0857 1111 2222", 1, "1.3", false);
        w.addButton.doClick();
        w.report("6. Ayu, Iron Only, 1.3 kg");
        System.out.println("   list:");
        for (int i = 0; i < w.listModel.size(); i++) {
            System.out.println("     " + w.listModel.get(i));
        }
        System.out.println("   " + w.totalLabel.getText());
        w.orderList.setSelectedIndex(1);
        System.out.println("   remove enabled after selecting row 2: "
```

Listing 10.19 (Continued) *LaundryOrderWindow.java: the Profitina Laundry order-entry window in Swing, with its model classes and a --selftest mode.*

```
                + w.removeButton.isEnabled());
        w.removeButton.doClick();
        w.report("7. Remove the selected order");
        for (int i = 0; i < w.listModel.size(); i++) {
            System.out.println("     " + w.listModel.get(i));
        }
        System.out.println("   " + w.totalLabel.getText());
        System.out.println("   remove enabled: " + w.removeButton.isEnabled());
    }
}
```

Let us walk through the view, since the model uses only ideas from earlier chapters.

Building the window. The constructor uses a `BorderLayout` with 6-pixel gaps: the form at `PAGE_START`, the scrolling list at `CENTER` (so the list is what grows), the footer at `PAGE_END` — the tree of **Figure 10.3**. `buildForm` lays out the labeled fields with a loop over two parallel arrays, then places the check box, the price, a nested `FlowLayout` panel of buttons and the status label. The `JList` displays a `DefaultListModel<String>`: adding to or removing from the model updates the list on the screen automatically.

Connecting the events. All the wiring is in `wireEvents`, one line per reaction, which makes the window's behavior readable in one place:

- The weight field's **document** — the model behind every Swing text component — accepts a `DocumentListener`, which is told about every insertion and removal of text, keystroke by keystroke. The anonymous class must implement all three methods of the interface; each simply calls `updatePrice()`. (An action listener on the field would fire only on Enter, too late for a live price.)
- Choosing a service and ticking *Express* fire action events; lambdas route both to `updatePrice()`.
- *Add order*, and Enter in the weight field, call `addOrder()`; *Clear* calls `clearForm()`.
- A `ListSelectionListener` on the list enables *Remove selected* exactly when a row is selected; the button calls `removeSelected()`.

The handlers. `updatePrice()` parses the weight and formats `OrderForm.quote`'s answer, or shows a dash if `parseKg` throws. `addOrder()` wraps `OrderForm.build` in `try`/`catch`: on success it adds the line to the book and the list model, clears the form and reports in green; on failure it shows the message in red and uses the `FormError`'s `field` to put the cursor where the mistake is. Every handler takes microseconds, so the EDT is never blocked. `removeSelected()` removes the same index from the book and the list model, keeping them in step.

Starting. `main` follows Section 10.4: `invokeLater`, decorated frame, `EXIT_ON_CLOSE`, the panel as the content pane, `pack()`, and `setLocationRelativeTo(null)`, which centers the window on the screen.

10.12.5 The self-test

The self-test drives the real components through a scripted session: it types into fields with `setText`, chooses services with `setSelectedIndex`, ticks the box with `doClick()`, presses buttons with `doClick()`, and selects a row with `setSelectedIndex`. Every one of these fires the same listeners a user would fire. `java LaundryOrderWindow --selftest` prints:

```
== 1. Add with an empty form
   price:  Price: -
   status: Customer name is required.
== 2. Siti, Wash & Fold, 4,2 kg (typed, not yet added)
   price:  Price: Rp31,500 for 4.5 kg, ready in 48 h
   status: Customer name is required.
==    ...Add order clicked
   price:  Price: -
   status: Added ORD-0001 for Siti Aminah.
== 3. Budi, Wash, Dry & Iron, 2.6 kg, express (typed)
   price:  Price: Rp45,000 for 3.0 kg, ready in 18 h
   status: Added ORD-0001 for Siti Aminah.
==    ...Add order clicked
   price:  Price: -
   status: Added ORD-0002 for Budi Santoso.
== 4. Ayu with phone 12345
   price:  Price: Rp10,000 for 2.0 kg, ready in 24 h
   status: Phone must be 10 to 13 digits starting with 0.
== 5. Ayu with weight abc
   price:  Price: -
   status: Weight must be a number, e.g. 4.5
== 5. Ayu with weight 0
   price:  Price: -
   status: Weight must be more than 0 kg.
== 5. Ayu with weight 75
   price:  Price: -
   status: Weight must be at most 50 kg.
== 6. Ayu, Iron Only, 1.3 kg
   price:  Price: -
   status: Added ORD-0003 for Ayu Lestari.
   list:
     ORD-0001  Siti Aminah    WF       4.5 kg    Rp31,500  48 h
     ORD-0002  Budi Santoso   EX-WDI   3.0 kg    Rp45,000  18 h
     ORD-0003  Ayu Lestari    IR       2.0 kg    Rp10,000  24 h
   3 orders   Total: Rp86,500
   remove enabled after selecting row 2: true
== 7. Remove the selected order
   price:  Price: -
   status: Removed ORD-0002.
     ORD-0001  Siti Aminah    WF       4.5 kg    Rp31,500  48 h
     ORD-0003  Ayu Lestari    IR       2.0 kg    Rp10,000  24 h
   2 orders   Total: Rp41,500
   remove enabled: false
```

Follow it against the requirements. (1) An empty form fails on the first rule, the name. (2) As soon as Siti's details are typed — before *Add order* — the `DocumentListener` has already turned `4,2` into Rp31,500 for 4.5 kg; the status still shows the old error. *Add order* accepts the order, clears the form and reports success. (3) Budi's express order shows Rp45,000 and 18 h while being typed. (4) The phone `12345` is rejected, while the live price for Ayu is already shown. (5) Three bad weights get three different messages. (6) The corrected order becomes ORD-0003, and the total is Rp86,500. (7) Selecting row 2 enables *Remove selected*; removing ORD-0002 leaves $86{,}500 - 45{,}000 = 41{,}500$ and disables the button again.

10.12.6 The Tkinter program

Listing 10.20 is the same program in Tkinter. Its model is line for line the same design, in Python idioms: an `Enum` whose members carry tuples of values, frozen `dataclass`es for `Quote` and `OrderLine`, and plain functions `quote`, `parse_kg` and `build_order` in place of the static methods of `OrderForm`.

Listing 10.20 `laundry_order_window.py`: the same order-entry window in Tkinter, with the same model and the same self-test.

```
"""laundry_order_window.py: the Profitina Laundry order-entry window in Tkinter."""
import math
import re
import sys
import tkinter as tk
from dataclasses import dataclass
from enum import Enum
from tkinter import ttk

# ---------- The model: plain Python, no Tkinter ----------


def rupiah(amount):
    return f"Rp{amount:,}"


class LaundryService(Enum):
    WASH_FOLD = ("WF", "Wash & Fold", 7000, 3.0, 48)
    IRON_ONLY = ("IR", "Iron Only", 5000, 2.0, 24)
    WASH_DRY_IRON = ("WDI", "Wash, Dry & Iron", 10000, 3.0, 72)

    def __init__(self, code, label, rate, min_kg, hours):
        self.code = code
        self.label = label
        self.rate = rate      # rupiah per kilogram
        self.min_kg = min_kg  # minimum chargeable weight
        self.hours = hours    # normal turnaround

    def __str__(self):  # what the combo box shows
        return f"{self.label} ({rupiah(self.rate)}/kg)"


@dataclass(frozen=True)
class Quote:
    chargeable_kg: float
    price: int
    hours: int
```

Listing 10.20 (Continued) *laundry_order_window.py: the same order-entry window in Tkinter, with the same model and the same self-test.*

```
@dataclass(frozen=True)
class OrderLine:
    number: str
    customer: str
    phone: str
    service: LaundryService
    express: bool
    quote: Quote

    def code(self):
        return ("EX-" if self.express else "") + self.service.code

    def display(self):
        q = self.quote
        return (f"{self.number}  {self.customer:<14.14} {self.code():<6} "
                f"{q.chargeable_kg:5.1f} kg {rupiah(q.price):>11} {q.hours:3d} h")


class FormError(ValueError):  # a message for the user, and the field to fix
    def __init__(self, field, message):
        super().__init__(message)
        self.field = field


def quote(s, kg, express):
    half_kg = math.ceil(2 * max(kg, s.min_kg) - 1e-9)
    price = s.rate * half_kg // 2
    hours = s.hours
    if express:
        price = 3 * price // 2
        hours = max(6, hours // 4)
    return Quote(half_kg / 2, price, hours)


def parse_kg(text):
    t = text.strip().replace(",", ".")  # accept 4,2 as well as 4.2
    if not t:
        raise FormError("weight", "Weight is required.")
    if not re.fullmatch(r"\d+(\.\d+)?", t):
        raise FormError("weight", "Weight must be a number, e.g. 4.5")
    kg = float(t)
    if kg <= 0:
        raise FormError("weight", "Weight must be more than 0 kg.")
    if kg > 50:
        raise FormError("weight", "Weight must be at most 50 kg.")
    return kg
```

Listing 10.20 (Continued) *laundry_order_window.py: the same order-entry window in Tkinter, with the same model and the same self-test.*

```
def build_order(number, name, phone, s, kg_text, express):
    n = name.strip()
    if not n:
        raise FormError("name", "Customer name is required.")
    if len(n) > 40:
        raise FormError("name", "Customer name must be at most 40 characters.")
    p = re.sub(r"[\s-]", "", phone)
    if not re.fullmatch(r"0\d{9,12}", p):
        raise FormError("phone", "Phone must be 10 to 13 digits starting with 0.")
    kg = parse_kg(kg_text)
    return OrderLine(number, n, p, s, express, quote(s, kg, express))


class OrderBook:
    def __init__(self):
        self._lines = []
        self._next = 1

    def next_number(self):
        return f"ORD-{self._next:04d}"

    def add(self, line):
        self._lines.append(line)
        self._next += 1

    def remove(self, index):
        return self._lines.pop(index)

    def lines(self):
        return tuple(self._lines)

    def total(self):
        return sum(line.quote.price for line in self._lines)


# ---------- The view: Tkinter widgets and their callbacks ----------

ERROR = "#aa0000"
OK = "#006e00"
SERVICES = list(LaundryService)
```

Listing 10.20 (Continued) *laundry_order_window.py: the same order-entry window in Tkinter, with the same model and the same self-test.*

```
class LaundryOrderWindow(tk.Frame):
    def __init__(self, master):
        super().__init__(master, padx=8, pady=8)
        self.book = OrderBook()
        self.name_var = tk.StringVar()
        self.phone_var = tk.StringVar()
        self.service_var = tk.StringVar(value=str(SERVICES[0]))
        self.weight_var = tk.StringVar()
        self.express_var = tk.BooleanVar(value=False)
        self.build_form()
        box = tk.LabelFrame(self, text="Orders")
        box.pack(fill=tk.BOTH, expand=True, pady=6)
        self.order_list = tk.Listbox(box, height=6, width=58, font="TkFixedFont",
                                     selectmode=tk.SINGLE, exportselection=False)
        bar = tk.Scrollbar(box, command=self.order_list.yview)
        self.order_list.config(yscrollcommand=bar.set)
        bar.pack(side=tk.RIGHT, fill=tk.Y)
        self.order_list.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        footer = tk.Frame(self)
        footer.pack(fill=tk.X)
        self.remove_button = tk.Button(footer, text="Remove selected",
                                       command=self.remove_selected, state=tk.DISABLED)
        self.remove_button.pack(side=tk.LEFT)
        self.total_label = tk.Label(footer, text="0 orders   Total: Rp0")
        self.total_label.pack(side=tk.RIGHT)
        self.wire_events()

    def build_form(self):
        form = tk.Frame(self)
        form.pack(fill=tk.X)
        self.name_entry = tk.Entry(form, width=20, textvariable=self.name_var)
        self.phone_entry = tk.Entry(form, width=20, textvariable=self.phone_var)
        self.service_box = ttk.Combobox(form, width=28, state="readonly",
                                        textvariable=self.service_var,
                                        values=[str(s) for s in SERVICES])
        self.weight_entry = tk.Entry(form, width=8, textvariable=self.weight_var)
        labels = ["Customer", "Phone", "Service", "Weight (kg)"]
        fields = [self.name_entry, self.phone_entry, self.service_box, self.weight_entry]
        for row, (text, field) in enumerate(zip(labels, fields)):
            tk.Label(form, text=text).grid(row=row, column=0, sticky="w", padx=4, pady=2)
            field.grid(row=row, column=1, sticky="w", padx=4, pady=2)
        self.express_check = tk.Checkbutton(form, text="Express (+50%, 4x faster)",
                                            variable=self.express_var,
                                            command=self.update_price)
        self.express_check.grid(row=3, column=2, sticky="w")
        self.price_label = tk.Label(form, text="Price: -")
```

Listing 10.20 (Continued) *laundry_order_window.py: the same order-entry window in Tkinter, with the same model and the same self-test.*

```
        self.price_label.grid(row=4, column=1, columnspan=2, sticky="w", padx=4)
        buttons = tk.Frame(form)
        buttons.grid(row=5, column=1, columnspan=2, sticky="w", padx=4, pady=2)
        self.add_button = tk.Button(buttons, text="Add order", command=self.add_order)
        self.clear_button = tk.Button(buttons, text="Clear", command=self.clear_form)
        self.add_button.pack(side=tk.LEFT)
        self.clear_button.pack(side=tk.LEFT)
        self.status_label = tk.Label(form, text=" ")
        self.status_label.grid(row=6, column=0, columnspan=3, sticky="w", padx=4)

    def wire_events(self):
        # A variable trace fires on every change, like a DocumentListener.
        self.weight_var.trace_add("write", lambda *args: self.update_price())
        self.service_var.trace_add("write", lambda *args: self.update_price())
        self.weight_entry.bind("<Return>", lambda event: self.add_order())
        self.order_list.bind("<<ListboxSelect>>", self.on_select)

    def service(self):
        return SERVICES[[str(s) for s in SERVICES].index(self.service_var.get())]

    def update_price(self):
        try:
            kg = parse_kg(self.weight_var.get())
        except FormError:
            self.price_label.config(text="Price: -")
            return
        q = quote(self.service(), kg, self.express_var.get())
        self.price_label.config(
            text=f"Price: {rupiah(q.price)} for {q.chargeable_kg:.1f} kg, ready in
{q.hours} h")

    def add_order(self):
        try:
            line = build_order(self.book.next_number(), self.name_var.get(),
                               self.phone_var.get(), self.service(),
                               self.weight_var.get(), self.express_var.get())
        except FormError as e:
            self.say(ERROR, str(e))
            widget = {"name": self.name_entry, "phone": self.phone_entry}.get(
                e.field, self.weight_entry)
            widget.focus_set()
        else:
            self.book.add(line)
            self.order_list.insert(tk.END, line.display())
            self.clear_form()
            self.say(OK, f"Added {line.number} for {line.customer}.")
            self.name_entry.focus_set()
```

Listing 10.20 (Continued) *laundry_order_window.py: the same order-entry window in Tkinter, with the same model and the same self-test.*

```
        self.refresh_total()

    def clear_form(self):
        self.name_var.set("")
        self.phone_var.set("")
        self.service_var.set(str(SERVICES[0]))
        self.weight_var.set("")
        self.express_var.set(False)
        self.price_label.config(text="Price: -")
        self.say(OK, " ")

    def on_select(self, event):
        chosen = self.order_list.curselection()
        self.remove_button.config(state=tk.NORMAL if chosen else tk.DISABLED)

    def remove_selected(self):
        chosen = self.order_list.curselection()
        if not chosen:
            return
        i = chosen[0]
        gone = self.book.remove(i)
        self.order_list.delete(i)
        self.say(OK, f"Removed {gone.number}.")
        self.on_select(None)
        self.refresh_total()

    def say(self, color, message):
        self.status_label.config(fg=color, text=message)

    def refresh_total(self):
        n = len(self.book.lines())
        self.total_label.config(
            text=f"{n} {'order' if n == 1 else 'orders'}   Total:
{rupiah(self.book.total())}")


# ---------- Self-test: drive the real widgets without showing them ----------

def fill(w, name, phone, service, kg, express):
    w.name_var.set(name)
    w.phone_var.set(phone)
    w.service_box.current(service)
    w.weight_var.set(kg)
    if w.express_var.get() != express:
        w.express_check.invoke()
```

Listing 10.20 (Continued) *laundry_order_window.py: the same order-entry window in Tkinter, with the same model and the same self-test.*

```
def report(w, step):
    print("==", step)
    print("   price: ", w.price_label.cget("text"))
    print("   status:", w.status_label.cget("text"))


def self_test():
    root = tk.Tk()
    root.withdraw()
    w = LaundryOrderWindow(root)
    w.pack(fill=tk.BOTH, expand=True)
    w.add_button.invoke()
    report(w, "1. Add with an empty form")
    fill(w, "Siti Aminah", "0812-3456-7890", 0, "4,2", False)
    report(w, "2. Siti, Wash & Fold, 4,2 kg (typed, not yet added)")
    w.add_button.invoke()
    report(w, "   ...Add order clicked")
    fill(w, "Budi Santoso", "081298765432", 2, "2.6", True)
    report(w, "3. Budi, Wash, Dry & Iron, 2.6 kg, express (typed)")
    w.add_button.invoke()
    report(w, "   ...Add order clicked")
    fill(w, "Ayu Lestari", "12345", 1, "1.3", False)
    w.add_button.invoke()
    report(w, "4. Ayu with phone 12345")
    for kg in ("abc", "0", "75"):
        fill(w, "Ayu Lestari", "0857 1111 2222", 1, kg, False)
        w.add_button.invoke()
        report(w, f"5. Ayu with weight {kg}")
    fill(w, "Ayu Lestari", "0857 1111 2222", 1, "1.3", False)
    w.add_button.invoke()
    report(w, "6. Ayu, Iron Only, 1.3 kg")
    print("   list:")
    for row in w.order_list.get(0, tk.END):
        print("    ", row)
    print("  ", w.total_label.cget("text"))
    w.order_list.selection_set(1)
    w.on_select(None)  # what Tk calls after a click on row 2
    print("   remove enabled after selecting row 2:",
          str(w.remove_button.cget("state") == tk.NORMAL).lower())
    w.remove_button.invoke()
    report(w, "7. Remove the selected order")
    for row in w.order_list.get(0, tk.END):
        print("    ", row)
    print("  ", w.total_label.cget("text"))
    print("   remove enabled:", str(w.remove_button.cget("state") == tk.NORMAL).lower())
    root.destroy()
```

Listing 10.20 (Continued) *laundry_order_window.py: the same order-entry window in Tkinter, with the same model and the same self-test.*

```
def main():
    if "--selftest" in sys.argv:
        self_test()
        return
    root = tk.Tk()
    root.title("Profitina Laundry - Order Entry")
    LaundryOrderWindow(root).pack(fill=tk.BOTH, expand=True)
    root.mainloop()


if __name__ == "__main__":
    main()
```

`xvfb-run -a python3 laundry_order_window.py --selftest` prints *exactly* the same lines as the Java version, character for character; the book verified this with `diff`:

```
== 1. Add with an empty form
   price:  Price: -
   status: Customer name is required.
== 2. Siti, Wash & Fold, 4,2 kg (typed, not yet added)
   price:  Price: Rp31,500 for 4.5 kg, ready in 48 h
   status: Customer name is required.
==    ...Add order clicked
   price:  Price: -
   status: Added ORD-0001 for Siti Aminah.
== 3. Budi, Wash, Dry & Iron, 2.6 kg, express (typed)
   price:  Price: Rp45,000 for 3.0 kg, ready in 18 h
   status: Added ORD-0001 for Siti Aminah.
==    ...Add order clicked
   price:  Price: -
   status: Added ORD-0002 for Budi Santoso.
== 4. Ayu with phone 12345
   price:  Price: Rp10,000 for 2.0 kg, ready in 24 h
   status: Phone must be 10 to 13 digits starting with 0.
== 5. Ayu with weight abc
   price:  Price: -
   status: Weight must be a number, e.g. 4.5
== 5. Ayu with weight 0
   price:  Price: -
   status: Weight must be more than 0 kg.
== 5. Ayu with weight 75
   price:  Price: -
   status: Weight must be at most 50 kg.
== 6. Ayu, Iron Only, 1.3 kg
   price:  Price: -
   status: Added ORD-0003 for Ayu Lestari.
   list:
     ORD-0001  Siti Aminah    WF       4.5 kg    Rp31,500  48 h
     ORD-0002  Budi Santoso   EX-WDI   3.0 kg    Rp45,000  18 h
     ORD-0003  Ayu Lestari    IR       2.0 kg    Rp10,000  24 h
   3 orders   Total: Rp86,500
   remove enabled after selecting row 2: true
== 7. Remove the selected order
   price:  Price: -
   status: Removed ORD-0002.
     ORD-0001  Siti Aminah    WF       4.5 kg    Rp31,500  48 h
     ORD-0003  Ayu Lestari    IR       2.0 kg    Rp10,000  24 h
   2 orders   Total: Rp41,500
   remove enabled: false
```

A few Tkinter-specific points deserve a word.

- **Control variables drive the live price.** Every entry is connected to a `StringVar` with `textvariable=`, and the combo box to `service_var`. `weight_var.trace_add("write", …)` calls a function every time the variable is written — whether by the user typing or by the program calling `set` — which is Tkinter's counterpart of the `DocumentListener`. A trace callback receives three arguments that we do not need, hence `lambda *args: self.update_price()`.
- **The combo box stores text.** A `ttk.Combobox` holds strings, so `service()` finds the enum member whose `str()` equals the selected text. `state="readonly"` stops the user from typing a service that does not exist.
- **The list box.** A `Listbox` holds rows of text itself (there is no separate list model); `insert(tk.END, row)` appends, `delete(i)` removes, and `curselection()` returns a tuple of selected indices. `exportselection=False` keeps the selection when the user clicks into another field. The scroll bar is connected in both directions: the bar's `command` scrolls the list, and the list's `yscrollcommand` moves the bar.
- **Geometry.** The form uses `grid`, the list and footer use `pack`, and the two never share a parent, as Section 10.8.5 requires.
- **Errors use `else`.** In `add_order`, the `try`/`except`/`else` form keeps the success path out of the `try` block, so that an unexpected exception in the success code is not mistaken for a validation error.

10.13 Profitina in Practice

Profitina's products are web applications rather than desktop windows, but the browser runs exactly the model of this chapter. A web page has one GUI thread; JavaScript code registers listeners with `addEventListener`, the browser's event loop dispatches clicks and key presses to them, and a listener that computes for too long freezes the page, just as a slow `actionPerformed` freezes a Swing window. The ideas carry over one for one.

Optimistic posting is the thread rule in action. When a user posts in Profitina's real-time community, the composer clears at once and the post appears immediately; the network request runs in the background, and if sending fails, the text is restored so nothing is lost. That is the shape of **Listing 10.17**: the click handler updates the display and returns within milliseconds, the slow work (here, a network round trip) runs elsewhere, and its result comes back to the GUI thread to confirm or undo — like `done()` in a `SwingWorker`.

The live feed is a queue of events. The community feed is updated by **Server-Sent Events** (SSE): the server pushes a small message whenever something happens, and the page's event loop delivers each message to a listener that inserts the new post. The page never asks "is there anything new?" in a loop; it registers a listener and waits, which is precisely **Pseudocode 10.1** with the server as one more source of events.

The dashboard is a view over a model. Profitina's dashboard gathers notifications, followed channels, rank and activity with one request and shows each item as something clickable that leads to its section. The page's code renders what the server's model says, and every click is a small handler; the scoring and the rules live on the server, where they can be tested without any browser at all — the same separation as **Figure 10.20**. And in keeping with the house rule that the application advises and the user decides, a window like the laundry form shows the price and explains every rejection, but it is the person at the counter who presses *Add order*.

10.14 Common Mistakes, and How to Avoid Them

1. Doing slow work in a listener. Reading a file, calling a server or sleeping inside `actionPerformed` or a Tk callback freezes the whole window. *Avoid it:* keep handlers short; move slow work to a `SwingWorker` or a thread with a queue and `after` (Section 10.11).

2. Touching components from another thread. Setting a label's text from a worker thread may work a hundred times and corrupt the display on the hundred-and-first. Tkinter often refuses at once with `RuntimeError: main thread is not in main loop`. *Avoid it:* in Swing, change components only on the EDT (`done()`, `process()`, `invokeLater`); in Tkinter, only on the `mainloop` thread, via a queue polled with `after`.

3. Forgetting `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`. Closing the window only hides it; the program keeps running with no window, and the terminal seems to hang. *Avoid it:* write the line for every main window (Section 10.4).

4. Forgetting `setVisible(true)`, or calling it before adding components. Nothing appears, or components added afterwards do not show until the window is resized. *Avoid it:* build everything, then `pack()`, then `setVisible(true)`, in that order.

5. Drawing outside `paintComponent`. Drawing with `getGraphics()` in a listener seems to work, and the drawing vanishes the next time the window is repainted (Section 10.10.6). *Avoid it:* store what should be drawn in fields, draw it in `paintComponent`, and call `repaint()` after changing the fields.

6. Forgetting `super.paintComponent(g)`. Old drawings stay on the screen under new ones, or garbage appears. *Avoid it:* make it the first line of every `paintComponent`.

7. Calling `paintComponent` yourself. You have no `Graphics` object for the screen, and painting is the event loop's job. *Avoid it:* call `repaint()`, which requests a painting.

8. Expecting a listener to run where it is registered. `button.addActionListener(e -> total += 10)` does not add 10; it arranges for 10 to be added *each time the button is clicked*. *Avoid it:* read `add…Listener` and `bind` as "remember this for later".

9. The late-binding trap in Python loops. `command=lambda: self.bump(n)` inside a loop makes every button use the last value of `n` (Section 10.10.5). *Avoid it:* `lambda n=n: self.bump(n)` or `functools.partial(self.bump, n)`.

10. Calling the callback instead of passing it. In Tkinter, `Button(..., command=self.add_order())` *calls* `add_order` once while the window is being built and passes its result, `None`, as the command, so the button does nothing. *Avoid it:* pass the function without parentheses, `command=self.add_order`, or wrap a call with arguments in a lambda.

11. Mixing `pack` and `grid` in the same parent. Tk 8.6 raises `TclError: cannot use geometry manager grid inside … which already has slaves managed by pack`, and older versions simply hang. *Avoid it:* use one geometry manager per parent; nest frames when you need both.

12. Absolute positions. `setLayout(null)` with `setBounds`, or `place(x=…, y=…)`, produces a window that breaks with other fonts, languages or window sizes. *Avoid it:* use layout managers, nesting panels as needed (Section 10.8).

13. Putting the rules inside the listeners. A price formula or a validation rule written inside `actionPerformed` can be tested only by clicking. *Avoid it:* keep rules in model classes that import nothing from the GUI, and let the listener call them (Section 10.12.3).

10.15 Chapter Summary

- A GUI program is **event-driven**: it builds a window, registers **event handlers** (Java *listeners*, Python *callbacks*), and hands control to the toolkit's **event loop**, which waits for events and calls the handlers — **inversion of control**.
- Java's GUI rests on **AWT** (native windows, events, `Graphics`, layout managers) and **Swing** (lightweight `J…` components that draw themselves). Python's standard toolkit is **Tkinter**, a Python layer over **Tcl/Tk**.
- A window is a **tree** of components. A `JFrame` holds a **content pane**; panels (`JPanel`, `tk.Frame`) are containers, and every real window part is best written as a class that extends one.
- The **event loop** (**Pseudocode 10.1**) waits for an event, finds its target, calls the listeners registered for it, and repaints when idle. While a listener runs, nothing else happens.
- To listen in Java: **declare** (`implements ActionListener`), **register** (`addActionListener`), **define** (`actionPerformed`). In Tkinter: `command=` or `bind("<Sequence>", callback)`. The event object says what happened (`getSource()`, `getPoint()`; `event.x`, `event.widget`).
- **Layout managers** (`FlowLayout`, `BorderLayout`, `BoxLayout`, `GridBagLayout`) and **geometry managers** (`pack`, `grid`) place components by rules, so windows survive other fonts and sizes; build complex windows by nesting.
- **Painting**: override `paintComponent`, call `super.paintComponent(g)` first, draw from fields, and call `repaint()` to request an update (immediate mode). A Tk `Canvas` keeps items and redraws them itself (retained mode). Colors are RGB triples, 256^3 of them (Equation (10.1)).
- **Inner classes** keep a hidden reference to their outer object; **anonymous classes** and **adapters** implement listeners in place; **lambdas** and **method references** are the short form for single-method interfaces and capture only **effectively final** locals, by copy. Python **closures** capture variables through cells and bind **late**; fix loops with `lambda n=n:` or `partial`.
- **The thread rule**: touch the GUI only on its own thread (Swing's EDT, Tk's `mainloop` thread) and never block that thread. Use `invokeLater`/`invokeAndWait` and `SwingWorker` in Swing; a worker thread, a `queue.Queue` and `after` in Tkinter.
- A **self-test mode** (`--selftest`) builds and drives the real components without showing them: headless mode in Swing, a withdrawn root under Xvfb in Tkinter.
- The laundry order-entry window separates a toolkit-free **model** (`LaundryService`, `Quote`, `OrderLine`, `FormError`, `OrderForm`, `OrderBook`) from the **view**; its Swing and Tkinter versions produce identical self-test output.

10.16 Exercises

Exercise 10.1 (tracing the event loop). In **Listing 10.3**, add a second listener to the button, registered *after* the first, that prints `button says: clicked`. Predict the complete output before running the program. Then move the line `queue.add(new Event("totalLabel", …))` so that the repaint event is queued *before* `total` is increased. What does the "screen" show now, and what general rule about listeners and repaint requests does this illustrate?

Exercise 10.2 (BorderLayout by hand). A panel of 400 × 200 pixels has a `BorderLayout`. Its `PAGE_START` component prefers a height of 30, its `PAGE_END` component a height of 20, its `LINE_START` component a width of 80 and its `LINE_END` component a width of 60; the `CENTER` component prefers 10 × 10. Compute the bounds (x, y, w, h) of all five components. Check your answer with the `report` method of **Listing 10.9**.

Exercise 10.3 (measure, don't assume). The five buttons of **Listing 10.9** have preferred widths 94, 94, 94, 187 and 42, and `FlowLayout` leaves a gap of 5 pixels between components and at both ends of a row. (a) Compute the smallest panel width at which, according to that description, all five fit in one row. (b) Find the real smallest width by experiment, changing the width passed to `report`. (c) Explain the difference.

Exercise 10.4 (a log of lines). Change **Listing 10.5** and **Listing 10.6** so that each Enter appends the typed text as a new line of a log (a `JTextArea` in Swing, a `Text` widget or a `Listbox` in Tkinter), clears the field for the next line, and ignores empty input. Extend the self-tests to type three lines, one of them empty, and print the log.

Exercise 10.5 (an oval where you click). Write, in both languages, a panel that shows the label *Click to draw an oval* and draws a circle of diameter 20 *centered* on the point where the user last clicked, as in **Figure 10.21**. Before the first click, the circle is centered at (50,50).

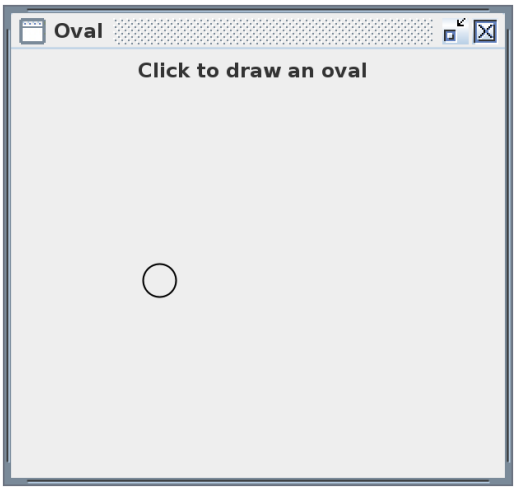


Figure 10.21 The oval program after a click: a small circle is drawn around the point that was clicked.

Exercise 10.6 (clamping). Using Equation (10.2) and the reasoning of Equation (10.3), how many clicks on *Darken* take the box of **Listing 10.11** from white to black if the step is 15 instead of 10? If it is 7? For which steps between 1 and 20 is the last click *not* clamped, that is, lands exactly on 0?

Exercise 10.7 (from `getSource` to lambdas). Rewrite **Listing 10.11** so that the panel no longer implements any listener interface: each button gets its own lambda, and the mouse click is handled by an anonymous `MouseAdapter`. Which methods disappear from the class? Which methods of the old class could other classes call that they now cannot?

Exercise 10.8 (late binding). Predict what this Python program prints, explain why, and fix it in two different ways.

```
fs = [lambda: i * 10 for i in range(3)]
print([f() for f in fs])
```

Exercise 10.9 (effectively final). `count` is a field of the class; the other variables are locals of the method `wire`. Which of the lines (1)–(5) compile? For each one that does not, explain the reason in one sentence.

```
int a = 1;
int b = 2;
b = 3;
final int[] c = {0};
String d = "x";
button.addActionListener(e -> System.out.println(a));  // (1)
button.addActionListener(e -> System.out.println(b));  // (2)
button.addActionListener(e -> c[0]++);                 // (3)
button.addActionListener(e -> d = d + "!");            // (4)
button.addActionListener(e -> count++);                // (5)
```

Exercise 10.10 (the thread rule). The following listener is meant to download today's price list and show how many services it contains. Find the two violations of the thread rule, explain what the user would experience because of each, and correct the code.

```
refresh.addActionListener(e -> {
    status.setText("Downloading...");
    String text = download("https://example.com/prices.csv");  // takes 2-5 s
    new Thread(() -> status.setText(text.lines().count() + " services")).start();
});
```

Exercise 10.11 (a board of machines). The laundry has four washing machines, M1–M4. Write a panel, in both languages, that draws them as four circles of diameter 60 in a row, green when free and red when busy, with their names underneath; a click inside a circle toggles that machine, and a label above shows *Busy: k of 4*. Clicks that are inside a circle's bounding square but outside the circle must be ignored. Give each program a `--selftest`.

Exercise 10.12 (members). Members get 10% off, computed as in Chapter 7: a price P becomes $P - \lfloor P/10 \rfloor$. Add a *Member* check box to the laundry window so that the discount applies to the order being entered, the live price shows it, and the list marks member orders. Say precisely which model classes and which view methods change, in both languages, and compute the price of Siti's order of **Figure 10.18** as a member.

Exercise 10.13 (testing the model without a window). Write a `unittest` test class for the model of **Listing 10.20** with at least four tests: the minimum weight, the express rule, the decimal comma, and a validation error that names the right field. Show that the tests run on a machine without a display, and explain why that is possible even though the module imports `tkinter`.

10.17 Solutions to the Exercises

Solution 10.1. Listeners run in the order they were registered, so each click now prints the second listener's line right after the first listener has run, and the repaint event, queued by the first listener, is dispatched on the next pass of the loop:

```
loop: text -> weightField
loop: click -> addButton
    button says: clicked
loop: repaint -> totalLabel
    screen shows Rp28000
loop: text -> weightField
loop: click -> addButton
    button says: clicked
loop: repaint -> totalLabel
    screen shows Rp49000
loop: close -> window
loop ended; total = Rp49000
```

If the repaint event is queued before the total is increased, the event carries the text `"Rp" + total` computed when it was created — the *old* total — and the screen is always one click behind:

```
loop: text -> weightField
loop: click -> addButton
loop: repaint -> totalLabel
    screen shows Rp0
loop: text -> weightField
loop: click -> addButton
loop: repaint -> totalLabel
    screen shows Rp28000
loop: close -> window
loop ended; total = Rp49000
```

In a real toolkit this mistake cannot happen, because `repaint()` carries no data: `paintComponent` reads the fields *when it runs*, after the listener has finished, so the order of `repaint()` and the field update inside one listener does not matter. The rule: a repaint request should say *that* something changed, not *what* it changed to; the painting code reads the current state.

Solution 10.2. The edges are placed first. `PAGE_START`: (0,0,400,30), the full width at its preferred height. `PAGE_END`: (0,180,400,20), at the bottom. The height between them is $200 - 30 - 20 = 150$. `LINE_START`: (0,30,80,150). `LINE_END`: $(400 - 60,30,60,150) = (340,30,60,150)$. `CENTER`: everything left, $(80,30,400 - 80 - 60,150) = (80,30,260,150)$; its preferred size of 10×10 is ignored. Running `report` on such a panel prints:

```
Border in 400 x 200:
  T                              x=  0 y=  0 w=400 h= 30
  B                              x=  0 y=180 w=400 h= 20
  L                              x=  0 y= 30 w= 80 h=150
  R                              x=340 y= 30 w= 60 h=150
  C                              x= 80 y= 30 w=260 h=150
```

Solution 10.3. (a) The row is $94 + 94 + 94 + 187 + 42 = 511$ pixels of buttons plus four gaps between them, $4 \times 5 = 20$, that is 531, plus a gap of 5 at each end: $531 + 10 = 541$. (b) The experiment says otherwise: at a width of 536 all five buttons still fit (the row starts at $x = 3$), and only at 535 does *5* move to a second row:

```
FlowLayout in 536 x 70:
  Button 1                       x=  3 y=  5 w= 94 h= 25
  Button 2                       x=102 y=  5 w= 94 h= 25
  Button 3                       x=201 y=  5 w= 94 h= 25
  Long-Named Button 4            x=300 y=  5 w=187 h= 25
  5                              x=492 y=  5 w= 42 h= 25
FlowLayout in 535 x 70:
  Button 1                       x= 25 y=  5 w= 94 h= 25
  Button 2                       x=124 y=  5 w= 94 h= 25
  Button 3                       x=223 y=  5 w= 94 h= 25
  Long-Named Button 4            x=322 y=  5 w=187 h= 25
  5                              x=246 y= 35 w= 42 h= 25
```

(c) `FlowLayout` subtracts the two end gaps from the width, $536 - 10 = 526$, and then, for each component, checks whether *the component alone* fits after the part of the row used so far — before adding the gap in front of it. After four buttons the row uses $94 + 5 + 94 + 5 + 94 + 5 + 187 = 484$, and $484 + 42 = 526$ fits, although with its gap the button needs 531. The row is then centered in 536 pixels, so the margins shrink from 5 to 3 and 2. The lesson is not the detail but the method: a layout manager's documentation describes its intent; only running it (as `report` does) tells you its exact arithmetic.

Solution 10.4. In Swing, add a field `private final JTextArea log = new JTextArea(5, 20);`, put it in a `JScrollPane` below the field, make it read-only with `log.setEditable(false)`, and change the listener:

```
    public void actionPerformed(ActionEvent e) {
        String s = t.getText().strip();
        if (!s.isEmpty()) {
            log.append(s + "\n");
        }
        t.setText("");
    }
```

In Tkinter, create `self.log = tk.Listbox(self, height=5)` and write

```
def on_enter(self, event):
    s = self.t.get().strip()
    if s:
        self.log.insert(tk.END, s)
    self.t.delete(0, tk.END)
```

The self-tests type each line with `setText` / `insert`, trigger the listener with `postActionEvent()` / `on_enter(None)`, and finally print `log.getText()` / `log.get(0, tk.END)`: two lines, the empty input having been ignored.

Solution 10.5. The state is the center (x, y); the painting code converts it to the bounding box that `drawOval` needs, whose top-left corner is 10 pixels (the radius) up and to the left of the center:

```
public class OvalClick extends JPanel {
    private int x = 50, y = 50;  // center of the circle

    public OvalClick() {
        setPreferredSize(new Dimension(350, 300));
        add(new JLabel("Click to draw an oval"));
        addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                x = e.getX();
                y = e.getY();
                repaint();
            }
        });
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawOval(x - 10, y - 10, 20, 20);  // bounding box around the center
    }
}
```

In Tkinter, the oval is an item created once and moved on each click:

```
self.oval = self.canvas.create_oval(40, 40, 60, 60)
self.canvas.bind("<Button-1>", self.on_click)

def on_click(self, event):
    self.canvas.coords(self.oval, event.x - 10, event.y - 10,
                       event.x + 10, event.y + 10)
```

A common mistake is `g.drawOval(e.getX(), e.getY(), 20, 20)`, which puts the *corner* of the bounding box at the click, so the circle appears below and to the right of the pointer.

Solution 10.6. From white (255) to black, the number of clicks with step s is $\lceil 255/s \rceil$. For $s = 15$: $255/15 = 17$ exactly, so 17 clicks, and the last one lands on 0 without clamping. For $s = 7$: $255/7 \approx 36.43$, so 37 clicks; after 36 the level is $255 - 252 = 3$, and the 37th is clamped from -4 to 0. The last click is unclamped exactly when s divides 255. Since $255 = 3 \times 5 \times 17$, the steps between 1 and 20 that divide it are 1, 3, 5, 15 and 17.

Solution 10.7. The constructor becomes

```
    darken.addActionListener(e -> shade(-10));
    lighten.addActionListener(e -> shade(+10));
    addMouseListener(new MouseAdapter() {
        @Override
        public void mouseClicked(MouseEvent e) {
            if (BOX.contains(e.getPoint())) {
                shade(+10);
            }
        }
    });
```

and the class header loses `implements ActionListener, MouseListener`. Six methods disappear: `actionPerformed` (with its `getSource` tests) and the five `MouseListener` methods, four of which were empty. Before the change, any class could call `panel.actionPerformed(...)` or `panel.mouseClicked(...)` and change the box without a real click, because interface methods are public; afterwards, the only way to change the box is through the buttons and the mouse. The listeners have become an implementation detail, which is what Chapter 7's encapsulation asks for.

Solution 10.8. It prints `[20, 20, 20]`. The list comprehension creates three lambdas, all of which refer to the same variable `i`; they look it up only when called, and by then the comprehension has finished with `i` equal to 2. Two fixes, both printing `[0, 10, 20]`:

```
fs = [lambda i=i: i * 10 for i in range(3)]                      # default argument
fs = [functools.partial(lambda i: i * 10, i) for i in range(3)]  # partial
```

A third, often clearest in real programs, is a factory function `def times_ten(i): return lambda: i * 10`, called as `times_ten(i)`: each call creates a new cell.

Solution 10.9. Lines (1), (3) and (5) compile; (2) and (4) do not:

```
Capture.java:13: error: local variables referenced from a lambda expression must be final
or effectively final
        button.addActionListener(e -> System.out.println(b));  // (2)
                                                         ^
Capture.java:15: error: local variables referenced from a lambda expression must be final
or effectively final
        button.addActionListener(e -> d = d + "!");            // (4)
                                       ^
Capture.java:15: error: local variables referenced from a lambda expression must be final
or effectively final
        button.addActionListener(e -> d = d + "!");            // (4)
                                      ^
3 errors
```

(1) `a` is assigned once: effectively final. (2) `b` is assigned twice, so it is not effectively final, even though the second assignment happens before the lambda. (3) `c` is final; the lambda changes an element of the array, not the variable. (4) The lambda itself assigns to `d`, which a lambda may never do to a captured local (the compiler reports both the read and the write). (5) `count` is a field, reached through the captured `this`, so it may be read and changed.

Solution 10.10. First violation: `download` runs inside the listener, on the EDT, for two to five seconds; the window freezes, and even the text *Downloading…* never appears, because the label can only be repainted after the listener returns. Second violation: the new thread calls `status.setText`, touching a Swing component off the EDT, which may corrupt the display now and then. The correct version downloads on a worker and updates the label in `done()`:

```
refresh.addActionListener(e -> {
    status.setText("Downloading...");
    refresh.setEnabled(false);
    new SwingWorker<String, Void>() {
        @Override
        protected String doInBackground() throws Exception {
            return download("https://example.com/prices.csv");
        }

        @Override
        protected void done() {
            try {
                status.setText(get().lines().count() + " services");
            } catch (Exception ex) {
                status.setText("Download failed: " + ex.getMessage());
            }
            refresh.setEnabled(true);
        }
    }.execute();
});
```

Disabling the button while the download runs prevents a second, overlapping download, and the `catch` turns a network failure into a message instead of a silent nothing.

Solution 10.11. **Listing 10.21** and **Listing 10.22** are the two programs. A point (x, y) lies inside the circle with center (c_x, c_y) and radius ρ exactly when its distance from the center is at most ρ; squaring both sides to avoid the square root gives the test

$$(x - c_x)^2 + (y - c_y)^2 \le \rho^2 \tag{10.8}$$

which `machineAt` applies to each machine in turn, with $\rho = 30$.

Listing 10.21 `MachineBoard.java`: four machines painted as circles; a click inside a circle, tested with Equation (10.8), toggles the machine.

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.image.BufferedImage;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class MachineBoard extends JPanel {
    static final int N = 4, SIZE = 60, GAP = 20, TOP = 40;
    private final boolean[] busy = new boolean[N];
    private final JLabel status = new JLabel();

    public MachineBoard() {
        setPreferredSize(new Dimension(N * (SIZE + GAP) + GAP, TOP + SIZE + 40));
        add(status);
        updateStatus();
        addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                int m = machineAt(e.getX(), e.getY());
                if (m >= 0) {
                    busy[m] = !busy[m];
                    updateStatus();
                    repaint();
                }
            }
        });
    }

    static int left(int m) {
        return GAP + m * (SIZE + GAP);
    }
```

Listing 10.21 (Continued) *MachineBoard.java: four machines painted as circles; a click inside a circle, tested with Equation (10.8), toggles the machine.*

```
    static int machineAt(int x, int y) {  // which circle contains (x, y)?
        for (int m = 0; m < N; m++) {
            double cx = left(m) + SIZE / 2.0, cy = TOP + SIZE / 2.0;
            double dx = x - cx, dy = y - cy;
            if (dx * dx + dy * dy <= (SIZE / 2.0) * (SIZE / 2.0)) {
                return m;
            }
        }
        return -1;
    }

    private void updateStatus() {
        int n = 0;
        for (boolean b : busy) {
            n += b ? 1 : 0;
        }
        status.setText("Busy: " + n + " of " + N);
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (int m = 0; m < N; m++) {
            g.setColor(busy[m] ? new Color(200, 40, 40) : new Color(40, 160, 70));
            g.fillOval(left(m), TOP, SIZE, SIZE);
            g.setColor(Color.BLACK);
            g.drawOval(left(m), TOP, SIZE, SIZE);
            g.drawString("M" + (m + 1), left(m) + SIZE / 2 - 8, TOP + SIZE + 16);
        }
    }

    public static void main(String[] args) throws Exception {
        if (args.length > 0 && args[0].equals("--selftest")) {
            System.setProperty("java.awt.headless", "true");
            SwingUtilities.invokeAndWait(MachineBoard::selfTest);
        } else {
            SwingUtilities.invokeLater(() -> {
                JFrame.setDefaultLookAndFeelDecorated(true);
                JFrame frame = new JFrame("Machine board");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.add(new MachineBoard());
                frame.pack();
                frame.setVisible(true);
            });
        }
    }
```

Listing 10.21 (Continued) *MachineBoard.java: four machines painted as circles; a click inside a circle, tested with Equation (10.8), toggles the machine.*

```
    void click(int x, int y) {
        dispatchEvent(new MouseEvent(this, MouseEvent.MOUSE_CLICKED,
                System.currentTimeMillis(), 0, x, y, 1, false, MouseEvent.BUTTON1));
    }

    static void selfTest() {
        MachineBoard b = new MachineBoard();
        System.out.println("start:             " + b.status.getText());
        b.click(50, 70);    // center of M1
        b.click(210, 70);   // center of M3
        System.out.println("click M1, M3:      " + b.status.getText());
        b.click(20, 40);    // inside M1's square but outside its circle
        System.out.println("click M1's corner: " + b.status.getText());
        b.click(50, 70);
        System.out.println("click M1 again:    " + b.status.getText());
        b.setSize(b.getPreferredSize());
        BufferedImage img = new BufferedImage(b.getWidth(), b.getHeight(),
                BufferedImage.TYPE_INT_RGB);
        b.paint(img.getGraphics());
        for (int m = 0; m < N; m++) {
            System.out.printf("M%d center pixel: #%06x%n", m + 1,
                    img.getRGB(left(m) + SIZE / 2, TOP + SIZE / 2) & 0xFFFFFF);
        }
    }
}
```

```
start:             Busy: 0 of 4
click M1, M3:      Busy: 2 of 4
click M1's corner: Busy: 2 of 4
click M1 again:    Busy: 1 of 4
M1 center pixel: #28a046
M2 center pixel: #28a046
M3 center pixel: #c82828
M4 center pixel: #28a046
```

The click at (20,40) is inside M1's bounding square, which runs from (20,40) to (80,100), but its distance from the center (50,70) is $\sqrt{30^2 + 30^2} \approx 42.4 > 30$, so Equation (10.8) rejects it and nothing changes. The pixels at the four centers confirm that only M3 is painted red at the end.

Listing 10.22 `machine_board.py`: the same board on a Tkinter canvas, where `tag_bind` on each oval does the hit test.

```
"""machine_board.py: click a washing machine to mark it busy or free."""
import sys
import tkinter as tk

N, SIZE, GAP, TOP = 4, 60, 20, 10
FREE, BUSY = "#28a046", "#c82828"


class MachineBoard(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.busy = [False] * N
        self.status = tk.Label(self)
        self.status.pack()
        self.canvas = tk.Canvas(self, width=N * (SIZE + GAP) + GAP,
                                height=TOP + SIZE + 30, highlightthickness=0)
        self.canvas.pack()
        self.circles = []
        for m in range(N):
            x = GAP + m * (SIZE + GAP)
            c = self.canvas.create_oval(x, TOP, x + SIZE, TOP + SIZE,
                                        fill=FREE, outline="black")
            self.canvas.create_text(x + SIZE / 2, TOP + SIZE + 14, text=f"M{m + 1}")
            # Tk tests "is the mouse over this oval?" for us.
            self.canvas.tag_bind(c, "<Button-1>", lambda event, m=m: self.toggle(m))
            self.circles.append(c)
        self.update_status()

    def toggle(self, m):
        self.busy[m] = not self.busy[m]
        self.canvas.itemconfig(self.circles[m], fill=BUSY if self.busy[m] else FREE)
        self.update_status()

    def update_status(self):
        self.status.config(text=f"Busy: {sum(self.busy)} of {N}")
```

Listing 10.22 (Continued) *machine_board.py: the same board on a Tkinter canvas, where tag_bind on each oval does the hit test.*

```
def self_test():
    root = tk.Tk()
    root.withdraw()
    b = MachineBoard(root)
    b.pack()
    print("start:            ", b.status.cget("text"))
    b.toggle(0)  # what Tk calls for a click on M1
    b.toggle(2)
    print("click M1, M3:     ", b.status.cget("text"))
    b.toggle(0)
    print("click M1 again:   ", b.status.cget("text"))
    for m, c in enumerate(b.circles):
        print(f"M{m + 1} fill: {b.canvas.itemcget(c, 'fill')}")
    root.destroy()


def main():
    if "--selftest" in sys.argv:
        self_test()
        return
    root = tk.Tk()
    root.title("Machine board")
    MachineBoard(root).pack(padx=6, pady=6)
    root.mainloop()


if __name__ == "__main__":
    main()
```

```
start:            Busy: 0 of 4
click M1, M3:     Busy: 2 of 4
click M1 again:   Busy: 1 of 4
M1 fill: #28a046
M2 fill: #28a046
M3 fill: #c82828
M4 fill: #28a046
```

In Tkinter, `tag_bind` on each oval makes Tk perform the hit test, and Tk tests the oval's true shape, not its bounding box, so corner clicks are ignored without any code of ours. The `lambda event, m=m: self.toggle(m)` uses the default-argument cure of Section 10.10.5; without `m=m`, every circle would toggle M4.

Solution 10.12. The model changes first, and the view follows.

- **Model.** `quote` gains a parameter `member` and, after the express rule, applies `price = price - price / 10` (Java, integer division) or `price -= price // 10` (Python). `OrderLine` gains a field `member`, and `display()` appends a marker such as `M` for member orders. `OrderForm.build` / `build_order` passes the flag through. Nothing else in the model changes, and `OrderBook.total()` needs no change at all, because it sums prices that already include the discount.
- **View.** Add `memberBox` (a `JCheckBox`) or `member_var` with a `Checkbutton`, next to *Express*; register `e -> updatePrice()` on it (Tkinter: `command=self.update_price`); pass `memberBox.isSelected()` to `quote` in `updatePrice` and to `build` in `addOrder`; reset it in `clearForm`.

Siti's order as a member: $P = 31{,}500$, discount $\lfloor 31{,}500/10 \rfloor = 3{,}150$, price 28,350. Note the order of the rules: for an express member order, the discount applies to the express price, $\lfloor 3P/2 \rfloor -$

$\lfloor \lfloor 3P/2 \rfloor / 10 \rfloor$, which is how Chapter 7's `Order` applied the member discount to a total that already contained express surcharges.

Solution 10.13. A test file `test_model.py` next to the program:

```
import unittest
from laundry_order_window import (FormError, LaundryService, build_order,
                                  parse_kg, quote)


class ModelTest(unittest.TestCase):
    def test_minimum_and_rounding(self):
        q = quote(LaundryService.IRON_ONLY, 1.3, False)
        self.assertEqual((q.chargeable_kg, q.price, q.hours), (2.0, 10000, 24))

    def test_express(self):
        q = quote(LaundryService.WASH_DRY_IRON, 2.6, True)
        self.assertEqual((q.price, q.hours), (45000, 18))

    def test_decimal_comma(self):
        self.assertEqual(parse_kg(" 4,2 "), 4.2)

    def test_bad_phone_names_the_field(self):
        with self.assertRaises(FormError) as ctx:
            build_order("ORD-0001", "Ayu", "12345", LaundryService.WASH_FOLD, "3", False)
        self.assertEqual(ctx.exception.field, "phone")


if __name__ == "__main__":
    unittest.main()
```

Run with the display variable removed (`env -u DISPLAY python3 test_model.py` on Linux), it reports:

```
Ran 4 tests in 0.000s

OK
```

Importing `laundry_order_window` imports `tkinter`, which only loads the library; a display is needed only when `tk.Tk()` is called, and the model functions never call it. This is the practical reward of Section 10.12.3: because the rules live in toolkit-free classes, they can be tested anywhere, quickly, with the tools of Chapter 6. The same holds in Java, where a JUnit test can call `OrderForm.quote` and `OrderForm.build` without any `java.awt.headless` setting at all.

References

[1] Oracle, "Creating a GUI with Swing," *The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/uiswing/

[2] OpenJFX project, "JavaFX 27 Highlights." [Online]. Available: https://openjfx.io/highlights/27/ (accessed Sep. 2026).

[3] Oracle, "The Arrival of Java 27," *Inside.java*, Sep. 15, 2026. [Online]. Available: https://inside.java/2026/09/15/jdk-27-available/

[4] OpenJDK, "JEP 398: Deprecate the Applet API for removal," 2021. [Online]. Available: https://openjdk.org/jeps/398

[5] P. Race, "JEP 504: Remove the Applet API," OpenJDK, 2025 (delivered in JDK 26, March 2026). [Online]. Available: https://openjdk.org/jeps/504

[6] J. K. Ousterhout and K. Jones, *Tcl and the Tk Toolkit*, 2nd ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2009.

[7] Python Software Foundation, “tkinter — Python interface to Tcl/Tk,” *The Python Standard Library*, version 3.14. [Online]. Available: https://docs.python.org/3/library/tkinter.html

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

[9] Oracle, “A visual guide to layout managers,” *The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

[10] A. Fowler, “Painting in AWT and Swing,” Sun Microsystems technical article, 2000. [Online]. Available: https://www.oracle.com/java/technologies/painting.html

[11] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*. Oracle America, 2023, §8.1.3 (inner classes), §15.9.5 (anonymous classes), §15.13 (method references), §15.27 (lambda expressions). [Online]. Available: https://docs.oracle.com/javase/specs/jls/se21/html/

[12] L. Ramalho, *Fluent Python*, 2nd ed. Sebastopol, CA, USA: O’Reilly Media, 2022, ch. 9.

[13] Python Software Foundation, “Why do lambdas defined in a loop with different values all return the same result?,” *Programming FAQ*. [Online]. Available: https://docs.python.org/3/faq/programming.html

[14] J. Bloch, *Effective Java*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2018, Items 24 and 42.

[15] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Upper Saddle River, NJ, USA: Addison-Wesley, 2006, ch. 9.

[16] Oracle, “Concurrency in Swing,” *The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/uiswing/concurrency/

[17] J. Nielsen, *Usability Engineering*. Boston, MA, USA: Academic Press, 1993, ch. 5.

[18] G. E. Krasner and S. T. Pope, “A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80,” *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26–49, 1988.

Chapter 11
The Model–View–Controller Pattern

By the end of this chapter you will be able to explain why a program with a window should be split into a **model** that holds the data and the rules, one or more **views** that show the data, and a **controller** that turns what the user does into operations on the model; draw the arrows between the three and say which of them may know about which; implement the **Observer** pattern that lets a model announce its changes without knowing who is listening, with `PropertyChangeSupport` in Java and a list of callables in Python; rebuild a small counter and a phone keypad with predictive text as clean MVC programs in Swing and in Tkinter; give every such program a `--selftest` mode that runs the model and the controller with no window at all; test a model with plain unit tests, never touching a button; and build a complete front-desk application for Profitina Laundry, with an order model, a form view, a list view, a summary view and a controller, together with its tests. You will also meet the two best-known descendants of MVC, Model–View–Presenter and Model–View–ViewModel, and know what each changes and why.

11.1 Where We Left Off

Chapter 10 gave our programs windows. We built frames and panels in Swing and in Tkinter, placed labels, text fields and buttons on them with layout managers, and learned the central idea of every graphical program: after the window has been built, the program no longer decides what happens next. It waits. The user clicks, types or closes the window, the toolkit turns that gesture into an *event*, and the toolkit calls a small piece of our code — a listener in Java, a callback in Python — that we registered in advance. We wrote those listeners as inner classes, as lambdas and as closures, and we saw that each one runs on the toolkit's single event thread, so it must finish quickly.

Chapter 10's programs were small, and it was natural to write each one as a single class: the class built the window, and its listeners did whatever the button was for. That style works beautifully for twenty lines. It stops working somewhere around two hundred, and this chapter begins by watching it stop. The cure is a way of dividing a graphical program into three kinds of object, each with one job. The division is called **Model–View–Controller**, MVC for short. It is more than forty years old, it is the ancestor of the way almost every desktop, web and phone application is organized today, and — the part that matters most to a working programmer — it is what makes a program with a window *testable*.

Underneath MVC lies a smaller idea that we meet on the way: the **Observer** pattern, in which one object keeps a list of other objects and tells all of them whenever it changes. You have used it already without the name: every `addActionListener` in Chapter 10 subscribed an observer to a button. In this chapter we write our own subjects, so that our *data* can tell our *windows* that something happened.

> ✓ **KEY INSIGHT.** A window is a poor place to keep the truth. Put the data and the rules in an object that knows nothing about windows (the model), let any number of windows draw that

> object (the views), and let a third object translate clicks into requests to the model (the controller). Everything in this chapter is an elaboration of that sentence.

11.2 A Story: The Front Desk That Knew Too Much

The front desk of Profitina Laundry has had a small program on its computer since the shop opened. A customer brings a bag of clothes; the desk clerk types the customer's name, chooses a service — Wash & Fold, Wash, Dry & Iron, or Iron Only — types the weight shown on the scale, ticks *Express* if the customer is in a hurry, and clicks **Add order**. The order appears in a list with its price. When the clothes move from the washer to the folding table, the clerk selects the order and clicks **Advance status**; later the order is *Ready*; finally the customer collects it and it is *Picked up*.

The first version of the program was written in one afternoon by a keen student, as one class called `FrontDeskWindow`. It was a subclass of `JFrame`. Its constructor built the text fields and the buttons. The listener of the **Add order** button did everything that adding an order means, and it looked roughly like this:

```
add.addActionListener(e -> {
    String name = customerField.getText().trim();
    if (name.isEmpty()) { statusLabel.setText("Name?"); return; }
    double kg = Double.parseDouble(weightField.getText());
    double rate = serviceBox.getSelectedIndex() == 0 ? 7000
                : serviceBox.getSelectedIndex() == 1 ? 10000 : 5000;
    long price = Math.round(Math.ceil(kg * 2) / 2 * rate);
    if (expressBox.isSelected()) price = price * 3 / 2;
    tableModel.addRow(new Object[] {++count, name, kg, price, "Received"});
    total += price;
    totalLabel.setText("Total: Rp" + total);
});
```

It worked. Then, over one year, four things happened.

First, the owner wanted a second screen. A television was mounted in the waiting area, and the owner wanted it to show the orders that were ready for pickup, in large letters, so that customers could see at a glance whether to queue. Where are the orders? They exist only as rows of a `DefaultTableModel` inside the front-desk window, as arrays of `Object`. The price is a number in a table cell; the status is a string in another. To show the ready orders elsewhere, the new screen must reach inside the old window and read its table cells — or the orders must be copied, and then the two copies drift apart the first time someone forgets to update one of them.

Second, the price rules changed. Each service got a minimum weight (three kilograms for washing, two for ironing), because washing half a kilogram costs the laundry almost as much water and electricity as washing three. The rule belongs to the business, but its code lived inside a button listener, next to `getText()` calls and table rows. The student who changed it had to read through widget code to find three lines of arithmetic, and a misplaced parenthesis in those three lines was discovered by a customer, not by the programmer.

Third, somebody asked for tests. "Can we check automatically that a 4.2 kg Wash & Fold still costs Rp31,500 after every change?" With the first design, the only way to compute a price was to build the whole window, type into its text fields and click its button. That needs a screen. The build server that runs the tests has no screen. So there were no tests, and every change was tested by hand, by clicking, when somebody remembered.

Fourth, the laundry wanted a web version for its second branch. None of the class could be reused, because every line of it was tangled with Swing.

None of these four stories is exotic. They are the ordinary life of any program that people actually use. And they have one cause: the class `FrontDeskWindow` did six different jobs, listed on the left of **Figure 11.1**. It built widgets, read what the user typed, checked the weight, computed the price, stored the orders, and redrew the list and the totals. Jobs that change for different reasons — the look of the window, the rules of the business, the storage of the data — were welded into one listener.

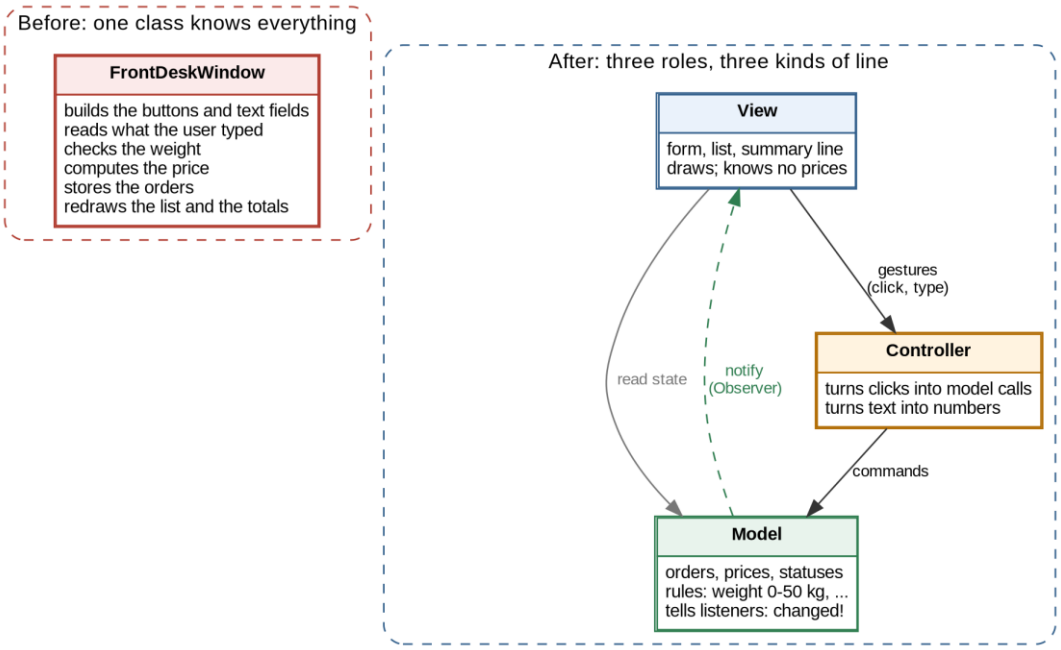


Figure 11.1 The same front desk before and after MVC. Left: one window class does six jobs. Right: the model holds the orders and the rules, the view draws, and the controller translates gestures into model calls; the dashed arrow is the Observer notification by which the model tells the view that something changed.

The right half of **Figure 11.1** shows the same program after it has been taken apart along the lines that MVC draws. The **model** holds the orders, their prices and statuses, and the rules about them, and it can tell anybody who is interested that it has changed. It does not import a single Swing or Tkinter class. The **view** draws: a form, a list, a line of totals. It contains no price rules at all; it shows whatever numbers the model reports. The **controller** listens to the view's buttons, turns the text the user typed into proper values, and calls the model. Each of the four stories now touches exactly one part: the television is a second view of the same model; the minimum-weight rule is a change to the model alone; the tests call the model directly, with no window; and the web version keeps the model and replaces the views and the controller. The rest of this chapter shows, in both languages, how to write programs that way.

11.3 Three Roles: Model, View, Controller

The MVC pattern separates the part of an application that holds and governs data from the part that presents it, and both from the part that handles the user's actions. The three roles are defined as follows.

> **DEFINITION.** The **model** represents the application's data and the rules that govern access to and updates of that data; in business software it is often an approximation, in code, of a real-world process. The **view** renders the contents of a model: it decides exactly how the data is

presented, and when the data changes, the view updates its presentation. The **controller** translates the user's interactions with the view into actions that the model performs.

Notice what each definition leaves *out*. The model's says nothing about pixels, fonts or buttons. The view's definition says nothing about what the data *means* — a view that shows an order's price does not know how the price was computed. The controller's definition says nothing about storing data or drawing it; the controller is a translator between two languages, the language of gestures ("the button labeled *Add order* was clicked while the weight field held `4,2`") and the language of the business ("add an order of 4.2 kg of Wash & Fold for Sari").

Figure 11.2 is the classic picture of the three roles and the messages between them, redrawn from the picture the Java documentation used when it described how to design Java applications with MVC [1]. Solid arrows are *method invocations*: one object calls a method of another. Dashed arrows are *events*: one object announces that something happened, without knowing in advance who will react.

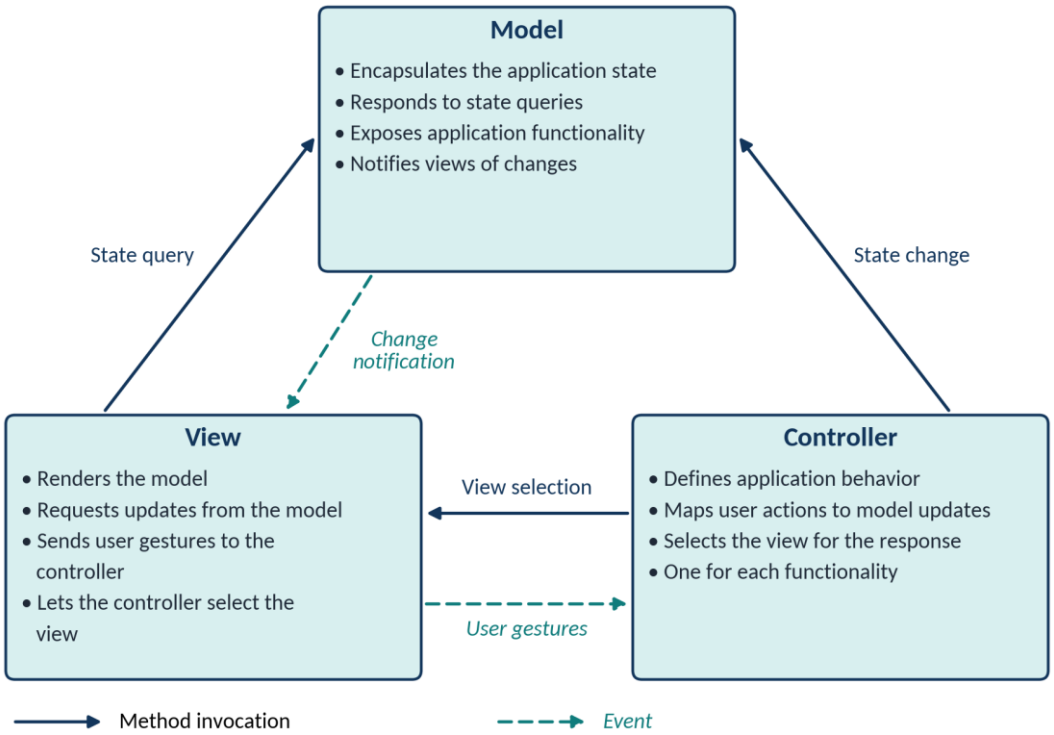


Figure 11.2 The classic MVC diagram. The model encapsulates the application state, answers state queries, exposes the application's functionality and notifies views of changes; the view renders the model and sends user gestures to the controller; the controller maps user actions to model updates and selects the view for the response.

Follow the arrows around the figure once, as if you were a single click. The user presses a button in the **view**; the view does not act on the click itself but passes it on as a *user gesture* to the **controller**. The controller decides what the gesture means and calls a method of the **model** that changes its state (*state change*). The model changes, and because views must stay current, it sends a *change notification* — an event — to every view that has registered interest. Each view then asks the model for the data it needs (*state query*) and redraws itself. The controller can also choose *which* view to show next (*view selection*), which matters in applications with several screens, such as a wizard or a web site.

The model in this cycle is passive in one important sense: it never calls a view by name. It only announces, "I have changed", to a list of listeners it was given. That is why the arrow from model

to view is dashed. The notification mechanism — the Observer pattern of Section 11.5 — is what lets the model stay ignorant of the views, and it is the single most important technical ingredient of MVC.

11.3.1 A model, a view and a controller you can point at

Figure 11.3 makes the three roles concrete for a tiny drawing program that places a line of semi-transparent text on a page. The model is five variables: an x position, a y position, an opacity, a font and the text itself. The view is a *Properties* window with a tab called *Text Element*, where those five values are displayed in text fields, a slider, a spinner and a text area. The controller is a set of methods, one per property: `changeXPosition`, `changeYPosition`, `changeOpacity`, `changeFont` and `changeText`.

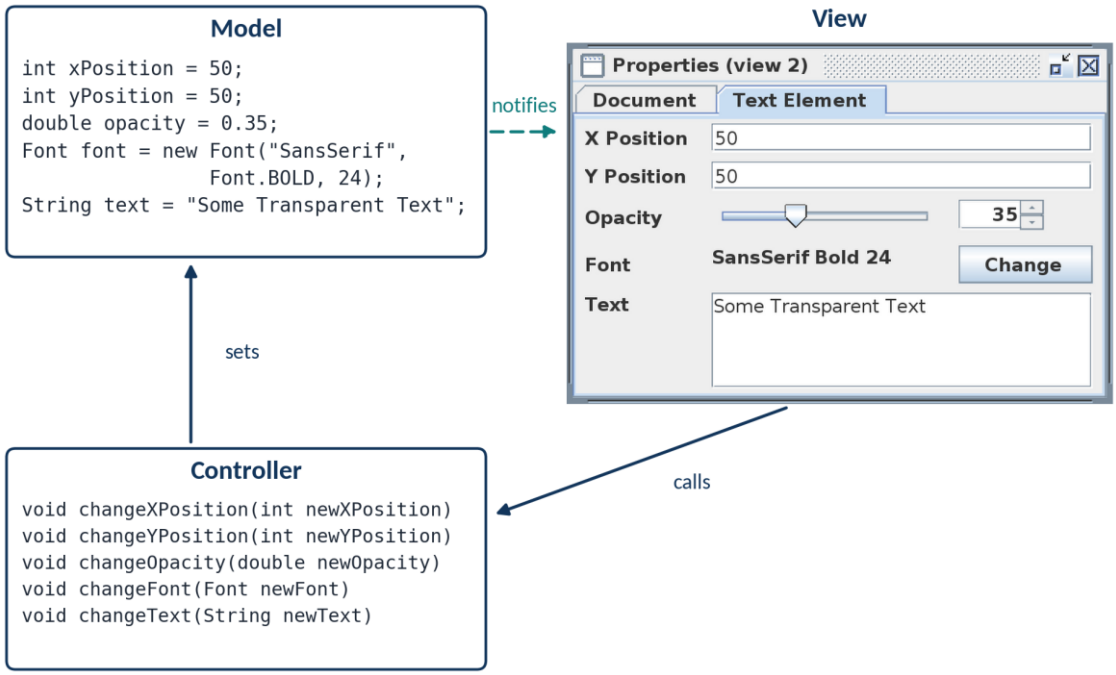


Figure 11.3 Model, view and controller of a small document editor. The model is five plain values; the view is a properties window that displays them; the controller offers one method per property, which the view calls when the user edits a field.

When the user drags the opacity slider from 35 to 50, the slider's listener calls `changeOpacity(0.50)`; the controller sets the model's opacity; the model notifies its views; and the view redraws the slider and the spinner at 50. Notice that the spinner next to the slider changes too, although the user never touched it. It changes *because the model changed*, not because the slider told it to. That is the first practical benefit of MVC: two widgets that show the same value can never disagree, because neither of them owns the value.

The second benefit is the one the laundry owner wanted: **the same model can have many views**. **Figure 11.4** adds a second window to the same five values. The *Display* window draws the text on a page, grey and semi-transparent, at position (50, 50); the *Properties* window shows the same values as numbers. Change the opacity in the properties window and the text on the page fades; type a new sentence in the text area and the page shows it. Neither window knows that the other exists. Each one only knows the model.

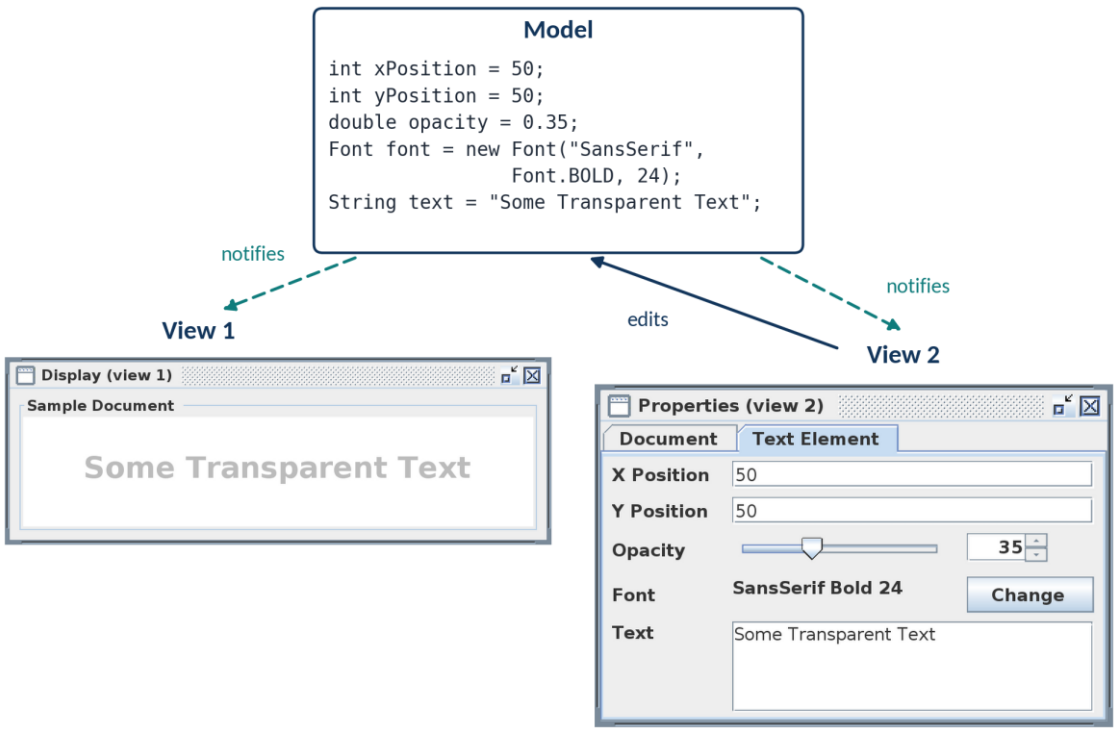


Figure 11.4 One model, two views. The *Display* window draws the text as it will appear on the page, and the *Properties* window shows the same values as numbers; both are kept current by the model's change notifications.

We can say the same thing with a small formula. At any moment t the model is in some state S_t — the values of all its fields. A view is, ideally, nothing but a function R (for *render*) from the model's state to what appears on the screen:

$$V_t = R(S_t) \tag{11.1}$$

Equation (11.1) is a design goal, and a useful one: the picture on the screen is *computed from* the model and holds no information of its own. Two views are two functions, R_1 and R_2, of the same S_t, so they cannot contradict each other; and a test of the model needs no view at all, because it checks S_t directly. When a program violates Equation (11.1) — when a text field holds a value that exists nowhere else — you have found the place where two screens will one day disagree.

11.3.2 Who may know whom

The three roles are only useful if the lines between them are respected. **Table 11.1** summarizes the rules. "Knows" here means "has a reference to, and calls methods of"; in Java it shows up as an `import` or a field type, in Python as an `import` or an attribute.

Table 11.1 The responsibilities and permitted dependencies of the three roles.

Role	Holds	May call	Must not know about
Model	the data and the business rules	nothing outside itself, except listeners it was given	views, controllers, the GUI toolkit
View	widgets, and how to draw the model	the model (read-only queries) or the controller	how the data is computed or stored
Controller	no data of its own, except wiring	the model (commands) and the view (updates)	how the view draws, how the model stores

The most important cell of **Table 11.1** is the last one of the first row: **the model must not know the GUI toolkit**. In Java, a model class must not import anything from `javax.swing` or `java.awt`. In Python, a model module must not import `tkinter`. This one rule is easy to check (search the file for the imports) and it guarantees three good things at once: the model can be tested without a screen, it can be reused under a different user interface, and it cannot accidentally depend on the order in which widgets happen to be redrawn.

The rules for the view and the controller are softer, and different programs draw them in different places. In the classic arrangement of **Figure 11.2**, the view holds a reference to the model and queries it directly when notified. In another common arrangement, shown in **Figure 11.5**, the controller sits in the middle as a **mediator**: the model's property-change events go to the controller, and the controller updates the view's components; the view reports user actions to the controller; the model and the view never meet. The figure's labels name the ingredients used in Java: the model is observed through property changes, and the view is built with Swing.

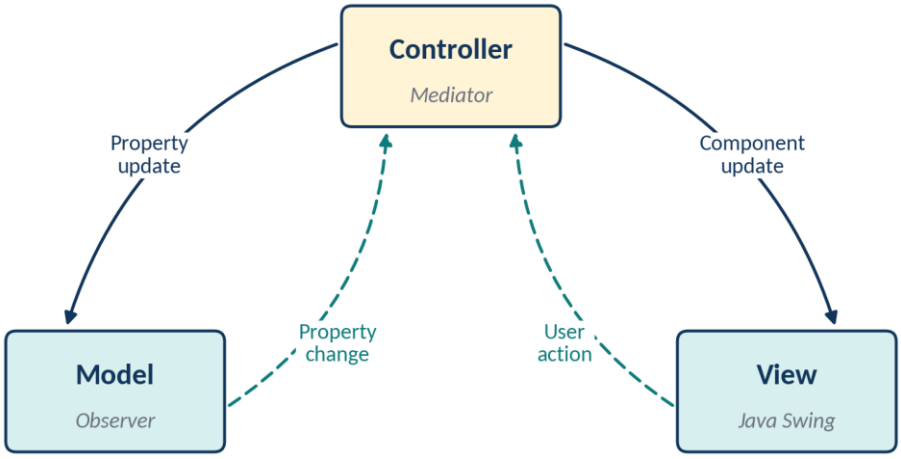


Figure 11.5 The controller as a mediator. User actions travel from the view to the controller, which updates the model's properties; the model's property changes travel back to the controller, which updates the view's components. The model and the view never refer to each other.

Both arrangements are MVC, and this chapter uses both: the phone keypad of Section 11.7 lets its views observe the model directly, and the laundry desk of Section 11.9 routes everything through a mediating controller, which makes the controller slightly larger and the views completely ignorant of the model.

11.3.3 Where the idea came from

MVC was described in 1979 by Trygve Reenskaug, a Norwegian computer scientist working at Xerox PARC on Smalltalk, the language in which windows, menus and the mouse were first combined in the way we now take for granted [2]. His problem was the one of our story: a user wants to see and manipulate the same information in several ways at once, and the program must keep all those ways consistent. His note called the parts *models*, *views* and *controllers*, and the idea was built into the Smalltalk-80 class library, where it was later described in a famous "cookbook" article [3]. The Observer pattern that connects models to views was one of the patterns cataloged in 1994 in *Design Patterns* by Gamma, Helm, Johnson and Vlissides [4], which also describes MVC as a combination of Observer with two other patterns.

In the original Smalltalk design the controller handled the raw mouse and keyboard; Swing and Tkinter already turn raw input into button clicks, so today's controllers receive "button *Add order*

clicked" rather than "mouse down at pixel (412, 37)". Swing itself is built on a variant of MVC: every `JTable` has a separate `TableModel`, every `JList` a `ListModel`, and every `JButton` a `ButtonModel`, while the view and the controller of each widget are merged into one "UI delegate" [5]. You already used this in Chapter 10 whenever you filled a `DefaultTableModel` rather than the table itself.

11.4 A First MVC Program: The Student Record

Before we meet windows, events and observers, let us see the three roles in the smallest program that can hold them. There is no GUI in it at all, which is a lesson in itself: MVC is about *responsibilities*, not about Swing. The example is a well-known one [6]: a student record with a name and a roll number (the number of the student in the class list), a view that prints the record on the console, and a controller that connects them. **Figure 11.6** is its class diagram.

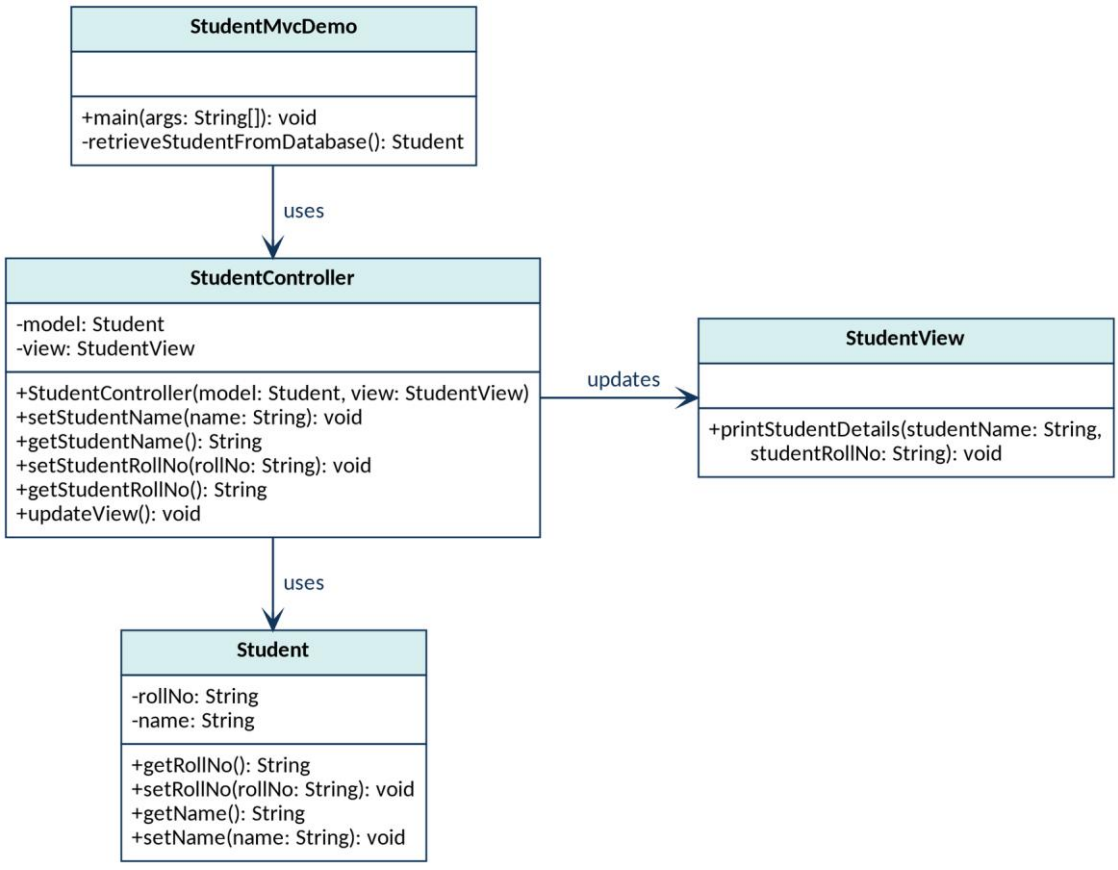


Figure 11.6 Class diagram of the student example. `StudentController` holds a `Student` model and a `StudentView`; it updates the model through setters and updates the view with `updateView()`; the demo class `StudentMvcDemo` uses the controller and retrieves the first record from a (simulated) database.

Read the diagram from the bottom. `Student`, the model, has two private fields, `rollNo` and `name`, each with a getter and a setter; nothing else. `StudentView` has one method, `printStudentDetails`, that prints what it is given. `StudentController` holds one model and one view (the `-` in front of `model` and `view` means *private*); it forwards `setStudentName` and the other setters to the model, and its method `updateView` asks the model for its values and hands them to the view. The arrows say who *uses* whom: the demo uses the controller, the controller uses the model and updates the view. There is no arrow from `Student` to anything. The model knows nobody.

Listing 11.1 is the program in Java, one file with the public class `StudentMvcDemo` and three package-private classes.

Listing 11.1 `StudentMvcDemo.java`: the smallest MVC program. `Student` is the model, `StudentView` the view and `StudentController` the controller.

```
public class StudentMvcDemo {
    public static void main(String[] args) {
        // Fetch the student record from the "database".
        Student model = retrieveStudentFromDatabase();
        // A view that writes the student's details on the console.
        StudentView view = new StudentView();
        StudentController controller = new StudentController(model, view);
        controller.updateView();
        // Change the model through the controller, then show it again.
        controller.setStudentName("John");
        controller.updateView();
    }

    private static Student retrieveStudentFromDatabase() {
        Student student = new Student();
        student.setName("Robert");
        student.setRollNo("10");
        return student;
    }
}

// Model: the data and the rules for reading and changing it.
class Student {
    private String rollNo;
    private String name;

    public String getRollNo() {
        return rollNo;
    }

    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Listing 11.1 (Continued) *StudentMvcDemo.java: the smallest MVC program. Student is the model, StudentView the view and StudentController the controller.*

```
// View: knows how to present a student, and nothing else.
class StudentView {
    public void printStudentDetails(String studentName, String studentRollNo) {
        System.out.println("Student: ");
        System.out.println("Name: " + studentName);
        System.out.println("Roll No: " + studentRollNo);
    }
}

// Controller: the only class that knows both the model and the view.
class StudentController {
    private final Student model;
    private final StudentView view;

    public StudentController(Student model, StudentView view) {
        this.model = model;
        this.view = view;
    }

    public void setStudentName(String name) {
        model.setName(name);
    }

    public String getStudentName() {
        return model.getName();
    }

    public void setStudentRollNo(String rollNo) {
        model.setRollNo(rollNo);
    }

    public String getStudentRollNo() {
        return model.getRollNo();
    }

    public void updateView() {
        view.printStudentDetails(model.getName(), model.getRollNo());
    }
}
```

Compile and run it in the chapter's `java` folder with `javac StudentMvcDemo.java` and `java StudentMvcDemo`:

```
Student:
Name: Robert
Roll No: 10
Student:
Name: John
Roll No: 10
```

Walk through `main` line by line, because every MVC program has the same four steps in the same order.

- **Build the model.** `retrieveStudentFromDatabase()` stands in for reading a record from a database; it creates a `Student`, sets its name to `"Robert"` and its roll number to `"10"`, and returns it. In a real program this would be a query; the point is that the model is created *first*, and independently of any view.
- **Build the view.** `new StudentView()` creates an object whose only skill is printing. It holds no student.

- **Build the controller, giving it both.** `new StudentController(model, view)` is the moment the three meet. The constructor stores the two references in `final` fields, so the wiring cannot be changed afterwards.
- **Let the controller drive.** `controller.updateView()` reads the model and tells the view to print; this produces the first three lines of output. `controller.setStudentName("John")` changes the model — and nothing else: no line is printed, because nobody asked the view to redraw. The second `updateView()` prints the new state, which is the last three lines.

That silent step in the middle deserves attention. After `setStudentName("John")`, the model says *John* while the last thing printed says *Robert*. For a moment, the view is out of date, and it stays out of date until somebody remembers to call `updateView()`. In this tiny program `main` remembers. In a real application, with dozens of places that change the model, somebody eventually forgets. The Observer pattern of the next section removes the need to remember: the model itself will announce every change.

(One small remark on the output: the first line of each block is `Student:` with a space after the colon, because the original `println("Student: ")` contains one. Invisible trailing spaces are a classic cause of failed output comparisons; the book reproduces the program's output exactly.)

Listing 11.2 is the same program in Python. The model uses properties (Chapter 7) instead of getter and setter methods, so the controller writes `self._model.name = name` where Java wrote `model.setName(name)`.

Listing 11.2 `student_mvc_demo.py`: the student record in Python, with the model's fields exposed as properties.

```
"""student_mvc_demo.py: the smallest MVC program, a student record on the console."""


class Student:
    """Model: the data and the rules for reading and changing it."""

    def __init__(self, name="", roll_no=""):
        self._name = name
        self._roll_no = roll_no

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

    @property
    def roll_no(self):
        return self._roll_no

    @roll_no.setter
    def roll_no(self, value):
        self._roll_no = value


class StudentView:
    """View: knows how to present a student, and nothing else."""

    def print_student_details(self, student_name, student_roll_no):
        print("Student: ")
        print("Name: " + student_name)
        print("Roll No: " + student_roll_no)


class StudentController:
    """Controller: the only class that knows both the model and the view."""

    def __init__(self, model, view):
        self._model = model
        self._view = view

    def set_student_name(self, name):
        self._model.name = name
```

Listing 11.2 (Continued) *student_mvc_demo.py: the student record in Python, with the model's fields exposed as properties.*

```
    def get_student_name(self):
        return self._model.name

    def set_student_roll_no(self, roll_no):
        self._model.roll_no = roll_no

    def get_student_roll_no(self):
        return self._model.roll_no

    def update_view(self):
        self._view.print_student_details(self._model.name, self._model.roll_no)


def retrieve_student_from_database():
    return Student(name="Robert", roll_no="10")


def main():
    # Fetch the student record from the "database".
    model = retrieve_student_from_database()
    # A view that writes the student's details on the console.
    view = StudentView()
    controller = StudentController(model, view)
    controller.update_view()
    # Change the model through the controller, then show it again.
    controller.set_student_name("John")
    controller.update_view()


if __name__ == "__main__":
    main()
```

Run it with `python3 student_mvc_demo.py` (on Windows, `python student_mvc_demo.py` or `py student_mvc_demo.py`). The output is identical to Java's:

```
Student:
Name: Robert
Roll No: 10
Student:
Name: John
Roll No: 10
```

Java's `private String name` with `getName` and `setName` becomes Python's `self._name` with a `@property` and a `@name.setter`; the leading underscore is Python's convention for "private" (Section 7.13). `retrieve_student_from_database` is a plain function, because Python does not require every function to live in a class.

Three classes to print six lines is more than a single `print` needs; the value appears the moment the program grows. Exercise 11.2 asks you to add a second view — one line of comma-separated values, ready for a spreadsheet — and you will find that you change the controller slightly, add one class, and do not touch the model at all. The model of a student does not care how many ways there are of printing it.

11.5 The Observer Pattern Underneath MVC

The student program had a weakness that we noted: after the model changed, the view stayed stale until somebody called `updateView()`. The Observer pattern cures it by turning the

relationship around. Instead of someone *telling the view* to redraw, the model *announces* that it has changed, and every interested party — every view, a log, a second screen — reacts.

> **DEFINITION.** In the **Observer** pattern, an object called the **subject** (or *observable*) keeps a list of **observers** (also called *listeners* or *subscribers*). Observers register with the subject by calling a method such as `addListener`, and may later unregister. Whenever the subject's state changes, it **notifies** every registered observer, usually by calling one agreed method on each. The subject knows only that its observers have that method; it does not know their classes or what they do [4].

The pattern is everywhere once you look for it. A Swing `JButton` is a subject whose observers are `ActionListener`s; a Tkinter `StringVar` is a subject whose observers are registered with `trace_add`; a newsletter is a subject whose observers are its subscribers. The subject's side of the contract is small: keep a list, offer *add* and *remove*, and call everybody on the list when something changes. The observer's side is smaller still: have the method the subject will call.

Figure 11.7 shows the pattern as we are about to write it in Java. The subject is a `WashCounter` that counts how many washing-machine loads the laundry has run today. It does not manage its observers itself; it *delegates* that job to a helper object from the standard library, `java.beans.PropertyChangeSupport`, which keeps the list and does the notifying. The observers implement the one-method interface `PropertyChangeListener`; ours are two lambdas, a display and a log.

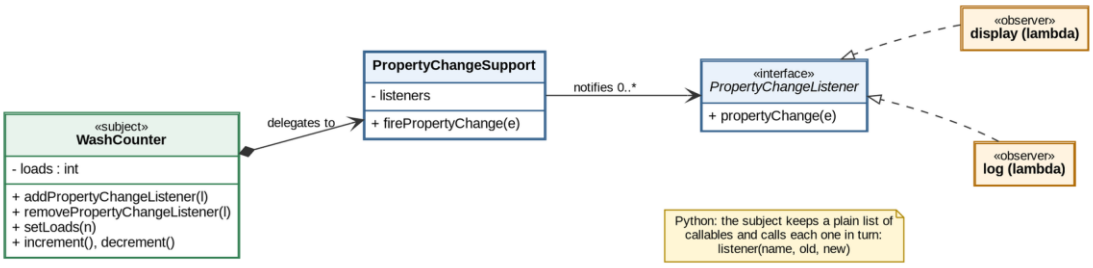


Figure 11.7 The Observer pattern as used by `ObserverDemo.java`. The subject `WashCounter` delegates its list of listeners to a `PropertyChangeSupport`, which notifies any number of objects implementing `PropertyChangeListener`; the two observers are lambdas. In Python the same roles are played by a plain list of callables.

11.5.1 Java: from `Observable` to `PropertyChangeSupport`

Java has offered two ready-made implementations of the subject side. The older one, the class `java.util.Observable` with its partner interface `java.util.Observer`, dates from Java 1.0 and is the one found in many older textbooks and course examples: a model `extends Observable`, calls `setChanged()` and `notifyObservers(value)` after every change, and a view `implements Observer` and receives `update(Observable o, Object arg)`. Since Java 9 both are **deprecated** [7]. The reasons are instructive. `Observable` is a *class*, so a model that extends it cannot extend anything else (Java has single inheritance, Chapter 7). Its `update` method passes the changed value as a bare `Object`, so every observer must cast it and hope. It says nothing about *which* property changed, so an observer of a model with ten fields cannot tell which one moved. And its notification order and thread-safety were never specified.

The recommended replacement is `java.beans.PropertyChangeSupport`, and it fixes each of those problems. It is used by *composition* — the model holds one as a field — so the model is free to extend any class it likes. Each notification is a `PropertyChangeEvent` that carries the source object, the *name* of the property that changed, its old value and its new value. And it has a helpful

built-in rule: if the old and the new value are equal, no event is fired at all. **Listing 11.3** puts it to work.

Listing 11.3 `ObserverDemo.java`: a subject, `WashCounter`, that announces every change of its `loads` property through `PropertyChangeSupport`, and two observers that know nothing of each other.

```
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class ObserverDemo {
    public static void main(String[] args) {
        WashCounter counter = new WashCounter();

        // Two observers. Neither knows the other exists.
        PropertyChangeListener display =
                e -> System.out.println("  [display] loads today: " + e.getNewValue());
        PropertyChangeListener log =
                e -> System.out.println("  [log] " + e.getPropertyName() + ": "
                        + e.getOldValue() + " -> " + e.getNewValue());

        counter.addPropertyChangeListener(display);
        counter.addPropertyChangeListener(log);

        System.out.println("increment()");
        counter.increment();
        System.out.println("increment()");
        counter.increment();
        System.out.println("setLoads(2)");
        counter.setLoads(2);  // same value: no event is fired
        counter.removePropertyChangeListener(log);
        System.out.println("decrement() after the log unsubscribed");
        counter.decrement();
        System.out.println("listeners now: " + counter.listenerCount());
        try {
            counter.setLoads(-1);
        } catch (IllegalArgumentException e) {
            System.out.println("setLoads(-1): " + e.getMessage());
        }
    }
}

// The subject: it keeps its data and announces every change.
class WashCounter {
    private final PropertyChangeSupport support = new PropertyChangeSupport(this);
    private int loads;

    void addPropertyChangeListener(PropertyChangeListener listener) {
        support.addPropertyChangeListener(listener);
    }
```

Listing 11.3 (Continued) *ObserverDemo.java: a subject, WashCounter, that announces every change of its loads property through PropertyChangeSupport, and two observers that know nothing of each other.*

```
    void removePropertyChangeListener(PropertyChangeListener listener) {
        support.removePropertyChangeListener(listener);
    }

    int listenerCount() {
        return support.getPropertyChangeListeners().length;
    }

    int getLoads() {
        return loads;
    }

    void setLoads(int newLoads) {
        if (newLoads < 0) {
            throw new IllegalArgumentException("loads cannot be negative");
        }
        int oldLoads = loads;
        loads = newLoads;
        // Fires only if the old and new values differ.
        support.firePropertyChange(
                new PropertyChangeEvent(this, "loads", oldLoads, newLoads));
    }

    void increment() {
        setLoads(loads + 1);
    }

    void decrement() {
        setLoads(loads - 1);
    }
}
```

Running `java ObserverDemo` prints:

```
increment()
  [display] loads today: 1
  [log] loads: 0 -> 1
increment()
  [display] loads today: 2
  [log] loads: 1 -> 2
setLoads(2)
decrement() after the log unsubscribed
  [display] loads today: 1
listeners now: 1
setLoads(-1): loads cannot be negative
```

Match the output to the code, event by event.

- The first `increment()` calls `setLoads(1)`. The method checks that the new value is not negative, remembers the old value 0, stores the new value 1, and calls `support.firePropertyChange(...)` with an event saying "*loads* went from 0 to 1". `PropertyChangeSupport` calls every registered listener in the order they registered: first `display`, which prints `[display] loads today: 1`, then `log`, which prints `[log] loads: 0 -> 1`. The counter itself printed nothing. It does not know that a display or a log exists.
- `setLoads(2)` is the interesting one: it prints *nothing* after its heading. The counter did call `firePropertyChange`, but the old and new values were both 2, and `PropertyChangeSupport` does not bother its listeners with a change that is not a change. This rule saves work and — more importantly — prevents a whole family of bugs in which two objects keep notifying each other of "changes" forever.

- `removePropertyChangeListener(log)` unsubscribes the log, so the following `decrement()` reaches only the display. The subject's list now holds one listener, as `listenerCount()` confirms.
- `setLoads(-1)` breaks the model's rule, and the model refuses with an `IllegalArgumentException` *before* changing anything or notifying anyone. A model that notifies first and validates afterwards would tell every view about a value that was about to be rejected.

Note that `increment` and `decrement` both go through `setLoads`, so there is exactly one place where the value changes and observers are told; make that a habit. And because `PropertyChangeListener` has a single abstract method, an observer can be a lambda, `e -> ...`, just like the `ActionListener`s of Chapter 10.

11.5.2 Python: an observer in a dozen lines

Python's standard library has no general-purpose observable class for plain objects — Tkinter's variables have one, but they need Tkinter — and it does not need one, because the pattern is a dozen lines of ordinary Python. A function is an object, so the subject can keep a list of *functions* and call each of them. **Listing 11.4** is the same counter.

Listing 11.4 `observer_demo.py`: the Observer pattern written by hand. The subject keeps a list of callables and calls each with the property name and the old and new values.

```
"""observer_demo.py: the Observer pattern written by hand in Python."""


class WashCounter:
    """The subject: it keeps its data and announces every change."""

    def __init__(self):
        self._loads = 0
        self._listeners = []

    def add_listener(self, listener):
        self._listeners.append(listener)

    def remove_listener(self, listener):
        self._listeners.remove(listener)

    def listener_count(self):
        return len(self._listeners)

    @property
    def loads(self):
        return self._loads

    def set_loads(self, new_loads):
        if new_loads < 0:
            raise ValueError("loads cannot be negative")
        old_loads = self._loads
        self._loads = new_loads
        if old_loads != new_loads:  # announce real changes only
            self._fire("loads", old_loads, new_loads)

    def increment(self):
        self.set_loads(self._loads + 1)

    def decrement(self):
        self.set_loads(self._loads - 1)

    def _fire(self, name, old, new):
        # Iterate over a copy: a listener may unsubscribe while being told.
        for listener in list(self._listeners):
            listener(name, old, new)


def main():
    counter = WashCounter()
```

Listing 11.4 (Continued) *observer_demo.py: the Observer pattern written by hand. The subject keeps a list of callables and calls each with the property name and the old and new values.*

```
    # Two observers. Neither knows the other exists.
    def display(name, old, new):
        print(f"  [display] loads today: {new}")

    def log(name, old, new):
        print(f"  [log] {name}: {old} -> {new}")

    counter.add_listener(display)
    counter.add_listener(log)

    print("increment()")
    counter.increment()
    print("increment()")
    counter.increment()
    print("set_loads(2)")
    counter.set_loads(2)  # same value: no event is fired
    counter.remove_listener(log)
    print("decrement() after the log unsubscribed")
    counter.decrement()
    print("listeners now:", counter.listener_count())
    try:
        counter.set_loads(-1)
    except ValueError as e:
        print("set_loads(-1):", e)


if __name__ == "__main__":
    main()
```

Run it with `python3 observer_demo.py`:

```
increment()
  [display] loads today: 1
  [log] loads: 0 -> 1
increment()
  [display] loads today: 2
  [log] loads: 1 -> 2
set_loads(2)
decrement() after the log unsubscribed
  [display] loads today: 1
listeners now: 1
set_loads(-1): loads cannot be negative
```

The output matches Java's line for line, apart from the method name `set_loads`, because the Python class implements the same contract by hand. `add_listener` and `remove_listener` manage a plain list. `set_loads` validates, stores, and calls `_fire` only if the value really changed — the rule that `PropertyChangeSupport` gave Java for free is here the single line `if old_loads != new_loads:`. And `_fire` loops over `list(self._listeners)`, a *copy* of the list, rather than the list itself. The comment says why: a listener may decide, while being notified, to unsubscribe itself (think of a one-time alert), and removing an element from a list while a `for` loop is walking through it silently skips the next element. Exercise 11.3 lets you watch that happen in both languages; the copy costs a few nanoseconds and removes the danger. (`PropertyChangeSupport` protects itself in the same way internally, which is one more reason to use it in Java instead of an `ArrayList` of your own.)

Table 11.2 puts the two implementations side by side.

Table 11.2 The Observer pattern in the two languages.

Ingredient	Java (`ObserverDemo.java`)	Python (`observer_demo.py`)
Observer type	interface `PropertyChangeListener`	any callable: function, lambda, bound method
List of observers	inside `PropertyChangeSupport`	`self._listeners`, a plain `list`
Subscribe / unsubscribe	`addPropertyChangeListener`, `removePropertyChangeListener`	`add_listener`, `remove_listener`
Notification	`firePropertyChange(event)`	`_fire(name, old, new)`
What an observer receives	a `PropertyChangeEvent`: source, name, old, new	the three arguments `name, old, new`
No event when nothing changed	built in	`if old_loads != new_loads:`
Safe if a listener unsubscribes itself	built in	iterate over `list(self._listeners)`

11.5.3 One change, many calls

Figure 11.8 draws the first `increment()` of **Listing 11.3** as a **sequence diagram**. A sequence diagram is the UML picture of *time*: each participating object has a vertical dashed *lifeline*, time flows downward, and each horizontal arrow is one message — a method call — numbered in the order it happens. A loop that returns to its own lifeline is a call of an object to itself; a dashed arrow is a return.

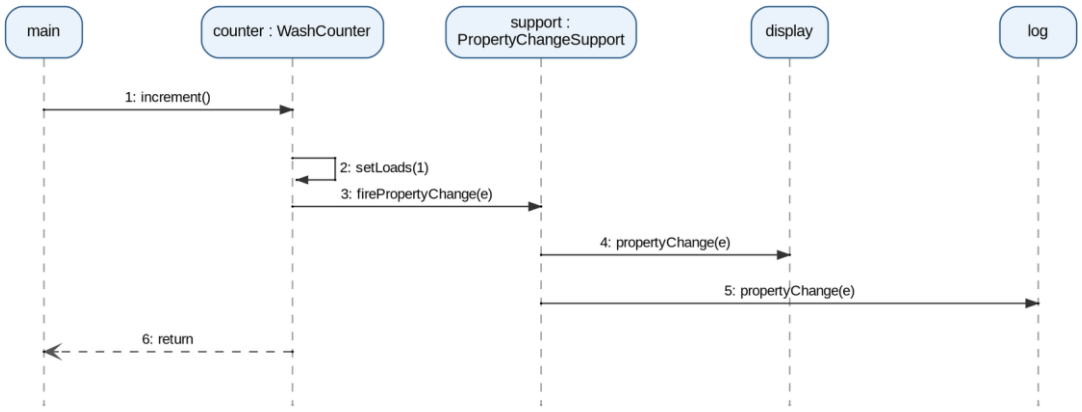


Figure 11.8 Sequence diagram of one `increment()`. The counter calls its own `setLoads(1)`, which asks the support object to fire one event; the support object calls each listener in registration order, and only then does `increment()` return to `main`.

Two facts can be read off the diagram. First, notification is **synchronous**: messages 4 and 5 happen *inside* message 1. `main`'s call to `increment()` does not return until every observer has finished reacting. If an observer is slow — if it redraws a large table, or writes to a disk — every change of the model becomes slow. Second, the number of calls grows with the number of observers. If k observers are registered and the model changes m times, the observers are called

$$C = m \cdot k \tag{11.2}$$

times in total. Equation (11.2) explains a design rule of every MVC program: *a model should notify once per logical change*, not once per field it touched. If adding an order changes the list of orders, the running total and the count of orders waiting, and the model fires one event for each, every

view redraws three times for one click — m has tripled for nothing. The laundry model of Section 11.9 therefore fires exactly one event, `orderBookChanged`, per operation, and each view recomputes everything it shows from the model's current state.

11.5.4 Push or pull?

The two programs above *push* data to their observers: every event carries the old and the new value, so an observer can react without asking the subject anything. The alternative is to *pull*: the event says only "I changed", and each observer then queries the subject for whatever it needs. The original counter example that Section 11.6 rebuilds used both styles: its `setValue` called `notifyObservers()` with nothing, while `incValue` called `notifyObservers(value)` with the new value.

Neither style is always better. Pushing is convenient when observers need exactly the value that changed, as our display does. Pulling is simpler when the model is large and different views need different parts of it: the model fires a single, content-free event, and each view reads what it needs — the list view reads the orders, the summary view reads the totals. Pulling also cannot deliver stale information, because each view reads the model's state *now*, after the whole change is complete. The keypad and the laundry desk later in this chapter pull; the counter in the next section pushes a single integer.

11.6 The Counter, Rebuilt: MVC with a Window

We now have the pieces to build a real, windowed MVC program. The first one is deliberately tiny: a counter with two buttons, **Up** and **Down**. It is the classic first example of MVC with a window, and **Figure 11.9** shows the original version at work. In the top row the program starts at 0; clicking **Up** three times produces 1, 2 and then, after a few more clicks, 10, and clicking **Down** repeatedly takes it to 5 and on down to −15, because nothing forbids negative values. In the bottom row the same program is started with the initial value 100 and goes up to 101 and down to 74.

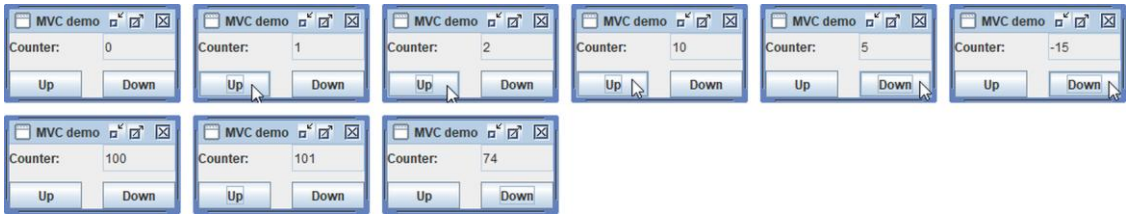


Figure 11.9 The Up/Down counter at work. Top row, started at 0: after clicks on *Up* the field shows 1, 2 and 10; after clicks on *Down* it shows 5 and −15. Bottom row, started at 100: 101, then 74. The mouse pointer shows which button was clicked last.

The original version of this program had four classes, in the style of the time: `Model extends Observable` held the integer; `MyPanel extends JPanel` built the label, the read-only text field and the two buttons in a `GridLayout` of two rows; `View implements Observer, ActionListener` created the frame, registered itself as the model's observer *and* as the listener of both buttons, and in `actionPerformed` called `model.incValue()` or `model.decValue()` depending on the button's action command; and `Controller` had a method `initView(title, value)` that asked the view to build the window. `Main` created the three and called `controller.initView("MVC demo", 0)`; changing the 0 to 100 gave the bottom row of **Figure 11.9**.

That version works, but look at where the jobs ended up. The class called `View` also handles the button clicks and calls the model — it is the controller as well as the view. The class called `Controller` does almost nothing. And the model extends a deprecated class. Our rebuilt version keeps the window exactly as it was and puts each job in its proper place.

- `CounterModel` holds the integer and a list of observers. It uses no GUI class.
- `CounterView` is an **interface**: the three things the controller needs from *any* view — show a value, tell me when *Up* is clicked, tell me when *Down* is clicked.
- `SwingCounterView` implements that interface with a real window. `TextCounterView` implements it by printing to the console, for the self-test.
- `CounterController` wires a model to a view: *Up* means `increment`, *Down* means `decrement`, and every change of the model is shown by the view.

Listing 11.5 is the Java program.

Listing 11.5 `CounterMvc.java`: the Up/Down counter as a model, a view interface with a Swing and a text implementation, and a controller. With `--selftest` it runs without a window.

```
import java.awt.GridLayout;
import java.util.ArrayList;
import java.util.List;
import java.util.function.IntConsumer;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class CounterMvc {
    public static void main(String[] args) {
        boolean selftest = false;
        int start = 0;
        for (String arg : args) {
            if (arg.equals("--selftest")) {
                selftest = true;
            } else {
                start = Integer.parseInt(arg);
            }
        }
        CounterModel model = new CounterModel(start);
        if (selftest) {
            selfTest(model);
            return;
        }
        // Swing objects are created on the event dispatch thread.
        SwingUtilities.invokeLater(() -> {
            SwingCounterView view = new SwingCounterView("MVC demo");
            new CounterController(model, view);
            view.open();
        });
    }
```

Listing 11.5 (Continued) *CounterMvc.java: the Up/Down counter as a model, a view interface with a Swing and a text implementation, and a controller. With --selftest it runs without a window.*

```
    // Model and controller at work, with a text view instead of a window.
    static void selfTest(CounterModel model) {
        TextCounterView view = new TextCounterView();
        new CounterController(model, view);
        int start = model.getValue();
        view.click("Up");
        view.click("Up");
        view.click("Down");
        view.click("Up");
        int expected = start + 2;
        if (model.getValue() != expected
                || !view.shownText().equals(Integer.toString(expected))) {
            throw new AssertionError("selftest failed");
        }
        System.out.println("selftest passed: model holds " + model.getValue()
                + ", view shows " + view.shownText());
    }
}

// Model: an integer and the rules for changing it. No Swing in sight.
class CounterModel {
    private final List<IntConsumer> listeners = new ArrayList<>();
    private int value;

    CounterModel(int value) {
        this.value = value;
    }

    int getValue() {
        return value;
    }

    void addListener(IntConsumer listener) {
        listeners.add(listener);
    }

    void increment() {
        setValue(value + 1);
    }

    void decrement() {
        setValue(value - 1);
    }
```

Listing 11.5 (Continued) *CounterMvc.java: the Up/Down counter as a model, a view interface with a Swing and a text implementation, and a controller. With --selftest it runs without a window.*

```
    void setValue(int newValue) {
        value = newValue;
        for (IntConsumer listener : listeners) {
            listener.accept(value);  // notify every observer
        }
    }
}

// What the controller needs from any view, window or not.
interface CounterView {
    void showValue(int value);

    void onUp(Runnable action);

    void onDown(Runnable action);
}

// Controller: turns user gestures into model operations and wires the parts.
class CounterController {
    CounterController(CounterModel model, CounterView view) {
        view.onUp(model::increment);
        view.onDown(model::decrement);
        model.addListener(view::showValue);
        view.showValue(model.getValue());  // the first picture
    }
}

// The real view: a small Swing window.
class SwingCounterView implements CounterView {
    private final JFrame frame;
    private final JTextField counter = new JTextField(6);
    private final JButton up = new JButton("Up");
    private final JButton down = new JButton("Down");

    SwingCounterView(String title) {
        frame = new JFrame(title);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel(new GridLayout(2, 2, 20, 10));
        panel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        counter.setEditable(false);
        panel.add(new JLabel("Counter:"));
        panel.add(counter);
        panel.add(up);
        panel.add(down);
        frame.setContentPane(panel);
    }
```

Listing 11.5 (Continued) *CounterMvc.java: the Up/Down counter as a model, a view interface with a Swing and a text implementation, and a controller. With --selftest it runs without a window.*

```
    void open() {
        frame.pack();
        frame.setLocationRelativeTo(null);  // center of the screen
        frame.setVisible(true);
    }

    @Override
    public void showValue(int value) {
        counter.setText(Integer.toString(value));
    }

    @Override
    public void onUp(Runnable action) {
        up.addActionListener(e -> action.run());
    }

    @Override
    public void onDown(Runnable action) {
        down.addActionListener(e -> action.run());
    }
}

// A stand-in view for the self-test: it prints instead of drawing.
class TextCounterView implements CounterView {
    private Runnable upAction = () -> { };
    private Runnable downAction = () -> { };
    private String shown = "";

    @Override
    public void showValue(int value) {
        shown = Integer.toString(value);
        System.out.println("  [view] Counter: " + shown);
    }

    @Override
    public void onUp(Runnable action) {
        upAction = action;
    }

    @Override
    public void onDown(Runnable action) {
        downAction = action;
    }
```

Listing 11.5 (Continued) *CounterMvc.java: the Up/Down counter as a model, a view interface with a Swing and a text implementation, and a controller. With --selftest it runs without a window.*

```
    void click(String button) {
        System.out.println("click " + button);
        if (button.equals("Up")) {
            upAction.run();
        } else {
            downAction.run();
        }
    }

    String shownText() {
        return shown;
    }
}
```

11.6.1 Reading the program

Start with `CounterModel`. Its list holds `IntConsumer`s — the standard functional interface for "something that accepts an `int`" — so an observer is anything with a method `void accept(int value)`. `setValue` stores the new value and calls `accept` on every observer: this is the push style, handing each observer the new number. `increment` and `decrement` go through `setValue`, so there is one place where the value changes and one place where observers are told. The model has no idea whether it is being shown in a window, printed on a console, or not shown at all.

`CounterView` is the contract between the controller and the views. It has three methods. `showValue(int)` is how a view *receives* data. `onUp(Runnable)` and `onDown(Runnable)` are how a view *reports gestures*: the controller hands the view a piece of code to run when the corresponding button is clicked. `Runnable` is the standard interface for "a piece of code that takes nothing and returns nothing", so `model::increment` (a method reference, Chapter 10) is a perfectly good `Runnable`.

`CounterController` is four lines long, and those four lines are the whole of the wiring. `view.onUp(model::increment)` says: when the user clicks *Up*, increment the model. `view.onDown(model::decrement)` says the same for *Down*. `model.addListener(view::showValue)` subscribes the view to the model, so every change of the model is shown. And `view.showValue(model.getValue())` draws the first picture, because the model has not changed yet and so has not notified anyone. Forgetting that last line is a common mistake: the window opens with an empty field, and the first click makes a number appear as if by magic.

`SwingCounterView` is the Swing code of Chapter 10: a `JPanel` with a 2-by-2 `GridLayout` holding a `JLabel`, a read-only `JTextField` and the two `JButton`s. Its `onUp` registers an `ActionListener` that simply runs the controller's action: the view does not decide what *Up* means.

`main` reads the command line. A number sets the starting value; the word `--selftest` selects the self-test. For a normal run it creates the view and the controller inside `SwingUtilities.invokeLater`, because Swing requires every widget to be created and touched on its single *event dispatch thread*, never on the `main` thread [8]; Chapter 10 explained why, and Chapter 14 returns to threads in depth.

11.6.2 The window

Run `java CounterMvc` on a computer with a screen. A small window titled **MVC demo** appears in the middle of the screen, about 190 by 80 pixels plus its title bar, arranged as a grid of two rows and two columns:

```
+---------------------------------+
| MVC demo                 _ o x |
+---------------------------------+
|  Counter:        [ 0         ] |
|                                 |
|  [   Up    ]     [   Down    ] |
+---------------------------------+
```

The top left cell is the label **Counter:**; the top right cell is a text field showing `0`, grey-backgrounded because it is not editable. The bottom row holds the buttons **Up** and **Down**, of equal width. Each click on **Up** adds one to the number in the field, each click on **Down** subtracts one, exactly as in **Figure 11.9**. `java CounterMvc 100` starts at 100. Closing the window ends the program, because of `EXIT_ON_CLOSE`.

11.6.3 The self-test

The computer that built this book has no screen, and neither has the server that runs the tests of most real projects. On such a machine, `java CounterMvc` fails the moment it tries to create a `JFrame` (Section 11.13 shows the exact error). This is where the separation pays off for the first time. Because the controller talks to the view only through the `CounterView` interface, we can hand it a *different* view — one that prints instead of drawing — and exercise the model and the controller completely, without a window. That is what `--selftest` does. `TextCounterView` stores the two actions it is given and offers a method `click(button)` that prints `click Up` or `click Down` and runs the corresponding action, just as Swing would when the real button is clicked. Its `showValue` prints what the real view would display.

Running `java CounterMvc --selftest` prints:

```
  [view] Counter: 0
click Up
  [view] Counter: 1
click Up
  [view] Counter: 2
click Down
  [view] Counter: 1
click Up
  [view] Counter: 2
selftest passed: model holds 2, view shows 2
```

The first line was printed by the controller's constructor, which drew the first picture. Each `click` then traveled the full MVC path: the view ran the controller's action, the action called the model, the model notified its observer, and the observer — the text view — printed the new value. At the end, `selfTest` checks that the model holds 2 and that the view shows 2; if either were wrong, it would throw an `AssertionError` and the program would end with an error. `java CounterMvc --selftest 100` runs the same clicks from 100 and ends with `selftest passed: model holds 102, view shows 102`.

What does the self-test prove, and what does it not? It proves that the model counts correctly, that the controller maps *Up* to increment and *Down* to decrement, that the observer is registered, and that the first picture is drawn. It does *not* prove that the Swing window looks right: that the label is spelled correctly, that the buttons are wide enough, or that the text field is read-only. Those facts live in `SwingCounterView`, a dozen lines that we deliberately kept free of logic so that there is little in them to get wrong. This is the general strategy for testing graphical programs: move every decision out of the view, test the decisions without a window, and keep the view so thin that reading it is enough [9].

Listing 11.6 is the same program in Python with Tkinter.

Listing 11.6 `counter_mvc.py`: the counter in Python. `TkCounterView` imports Tkinter inside its constructor, so the self-test runs even where Tkinter is not installed.

```
"""counter_mvc.py: the Up/Down counter as model, view and controller."""
import sys


class CounterModel:
    """Model: an integer and the rules for changing it. No Tkinter in sight."""

    def __init__(self, value=0):
        self._value = value
        self._listeners = []

    @property
    def value(self):
        return self._value

    def add_listener(self, listener):
        self._listeners.append(listener)

    def increment(self):
        self.set_value(self._value + 1)

    def decrement(self):
        self.set_value(self._value - 1)

    def set_value(self, new_value):
        self._value = new_value
        for listener in list(self._listeners):
            listener(self._value)  # notify every observer


class CounterController:
    """Controller: turns user gestures into model operations and wires the parts."""

    def __init__(self, model, view):
        view.on_up(model.increment)
        view.on_down(model.decrement)
        model.add_listener(view.show_value)
        view.show_value(model.value)  # the first picture


class TkCounterView:
    """The real view: a small Tkinter window."""
```

Listing 11.6 (Continued) *counter_mvc.py: the counter in Python. TkCounterView imports Tkinter inside its constructor, so the self-test runs even where Tkinter is not installed.*

```
    def __init__(self, title):
        import tkinter as tk  # imported here, so the self-test needs no Tkinter
        self._root = tk.Tk()
        self._root.title(title)
        self._text = tk.StringVar()
        tk.Label(self._root, text="Counter:").grid(
            row=0, column=0, sticky="w", padx=10, pady=5)
        tk.Entry(self._root, textvariable=self._text, width=6,
                 state="readonly").grid(row=0, column=1, padx=10, pady=5)
        self._up = tk.Button(self._root, text="Up", width=8)
        self._up.grid(row=1, column=0, padx=10, pady=5)
        self._down = tk.Button(self._root, text="Down", width=8)
        self._down.grid(row=1, column=1, padx=10, pady=5)

    def open(self):
        self._root.mainloop()

    def show_value(self, value):
        self._text.set(str(value))

    def on_up(self, action):
        self._up.config(command=action)

    def on_down(self, action):
        self._down.config(command=action)


class TextCounterView:
    """A stand-in view for the self-test: it prints instead of drawing."""

    def __init__(self):
        self._up = lambda: None
        self._down = lambda: None
        self.shown = ""

    def show_value(self, value):
        self.shown = str(value)
        print("  [view] Counter: " + self.shown)

    def on_up(self, action):
        self._up = action

    def on_down(self, action):
        self._down = action
```

Listing 11.6 (Continued) *counter_mvc.py: the counter in Python. TkCounterView imports Tkinter inside its constructor, so the self-test runs even where Tkinter is not installed.*

```
    def click(self, button):
        print("click " + button)
        if button == "Up":
            self._up()
        else:
            self._down()


def self_test(model):
    """Model and controller at work, with a text view instead of a window."""
    view = TextCounterView()
    CounterController(model, view)
    start = model.value
    view.click("Up")
    view.click("Up")
    view.click("Down")
    view.click("Up")
    expected = start + 2
    assert model.value == expected and view.shown == str(expected), "selftest failed"
    print(f"selftest passed: model holds {model.value}, view shows {view.shown}")


def main(args):
    selftest = "--selftest" in args
    numbers = [a for a in args if a != "--selftest"]
    model = CounterModel(int(numbers[0]) if numbers else 0)
    if selftest:
        self_test(model)
        return
    view = TkCounterView("MVC demo")
    CounterController(model, view)
    view.open()


if __name__ == "__main__":
    main(sys.argv[1:])
```

`python3 counter_mvc.py --selftest` prints exactly the same lines as the Java self-test:

```
  [view] Counter: 0
click Up
  [view] Counter: 1
click Up
  [view] Counter: 2
click Down
  [view] Counter: 1
click Up
  [view] Counter: 2
selftest passed: model holds 2, view shows 2
```

The structure is the Java structure, with Python's lighter syntax. The model's observers are any callables taking one argument. The controller passes bound methods — `model.increment`, `view.show_value` — which are Python's equivalent of Java's method references. There is no `CounterView` interface in the code, because Python checks at run time whether an object has the methods it is asked for (*duck typing*): anything with `show_value`, `on_up` and `on_down` is a counter view. (Section 11.9 shows how to *write down* such a contract with `typing.Protocol` when a program is large enough to need it.)

The Tkinter view [10] builds the same two-by-two grid with `grid(row=..., column=...)`. The number is shown in an `Entry` with `state="readonly"`, connected to a `StringVar`, so `show_value` only has to call `self._text.set(...)`. `on_up` sets the button's `command`, which is how Tkinter says

"call this when clicked". And `open()` calls `mainloop()`, which hands control to Tkinter's event loop until the window is closed.

One line in the Tkinter view is unusual: `import tkinter as tk` sits *inside* `__init__`, not at the top of the file. That way the model, the controller and the self-test run even on a machine where Tkinter is not installed — common on servers, and true of the machine that ran every program of this book. (On Linux, Tkinter is often a separate package, `python3-tk` on Debian and Ubuntu; the python.org installers for Windows and macOS include it.) Without it, `python3 counter_mvc.py` stops with `ModuleNotFoundError: No module named 'tkinter'`, but `python3 counter_mvc.py --selftest` runs. The Tkinter window has the same layout as the Swing one: **Counter:** and a read-only field showing `0` in the first row, **Up** and **Down** in the second.

11.7 The Phone Keypad: Predictive Text in MVC

Our second windowed program has a story of its own. Before touch screens, phones had twelve keys, and each key from 2 to 9 carried three or four letters: 2 for *abc*, 3 for *def*, up to 9 for *wxyz*. In the oldest scheme, *multi-tap*, you pressed 4 twice for *h*, 3 twice for *e*, and so on — thirteen presses for *hello*. In the 1990s a small company called Tegic Communications invented **T9**, "text on 9 keys", in which you press each key *once* per letter and the phone guesses the word from a dictionary [11]. *hello* becomes the five presses 4-3-5-5-6. When several words share the same keys — 4-6-6-3 is *good*, but also *home*, *gone*, *hood*, *hoof* and *hone* — the phone shows one and you press a *next* key, usually the star, to cycle through the others.

Our keypad program implements exactly that, and its design follows the original set of classes for this exercise: a model called `DictionaryModel` that held the message, a view `MessageView` that displayed it in a text area, a panel `KeypadController` with twelve buttons, a listener class `KeypadListener` that forwarded each button to the model, a `MessageComponent` that assembled the parts, and `MessageGUI` with the `main` method. The dictionary lookup itself came from a class called `PredictivePrototype`, which turned a word into its key sequence with a long chain of `if` statements. We keep that architecture and rebuild each class cleanly.

11.7.1 Words, keys and signatures

The model rests on one function. Let κ (the Greek letter *kappa*, for *key*) map each letter to its key: $\kappa(a) = \kappa(b) = \kappa(c) = 2$, $\kappa(d) = \kappa(e) = \kappa(f) = 3$, and so on, with $\kappa(p) = \kappa(q) = \kappa(r) = \kappa(s) = 7$ and $\kappa(w) = \kappa(x) = \kappa(y) = \kappa(z) = 9$. The **signature** of a word $w = c_1 c_2 \ldots c_n$ of n letters is the string of its keys:

$$\sigma(c_1 c_2 \ldots c_n) = \kappa(c_1)\,\kappa(c_2) \ldots \kappa(c_n) \tag{11.3}$$

For example, by Equation (11.3), $\sigma(\text{hello}) = \kappa(h)\kappa(e)\kappa(l)\kappa(l)\kappa(o) = 4\,3\,5\,5\,6$, that is, `43556`; $\sigma(\text{wash}) = 9\,2\,7\,4$; and $\sigma(\text{good}) = \sigma(\text{home}) = 4663$. The function σ is not one-to-one — different words can share a signature — and that is precisely why the user sometimes needs the star key. The dictionary's job is the inverse direction: given a signature, return every word that has it. If the dictionary holds N words, we compute every signature once when the dictionary is built and store the words in a map from signature to list of words; after that, finding the words for a signature is one map lookup, however large N is.

The model of the message then needs very little state: the words already accepted, the signature typed so far for the current word, the list of dictionary words that match that signature, and the

index of the match currently shown. Four operations change it, and the controller maps each key to one of them:

Table 11.3 What each key of the keypad does to the message model.

Key	Operation	Effect on the model
2 … 9	`addDigit(d)`	append *d* to the signature and look it up again; show the first match
*	`nextMatch()`	show the next match, wrapping round to the first after the last
0	`acceptWord()`	accept the word shown, as if a space had been typed, and start a new word
C	`deleteDigit()`	remove the last digit of the signature and look it up again
1	—	nothing: key 1 carries no letters

When the signature typed so far matches no word — as happens in the middle of *hello*, since no word in the dictionary has the signature `435` — the model shows the digits themselves, so the user can see that the keys were registered. **Figure 11.10** shows how the classes fit together.

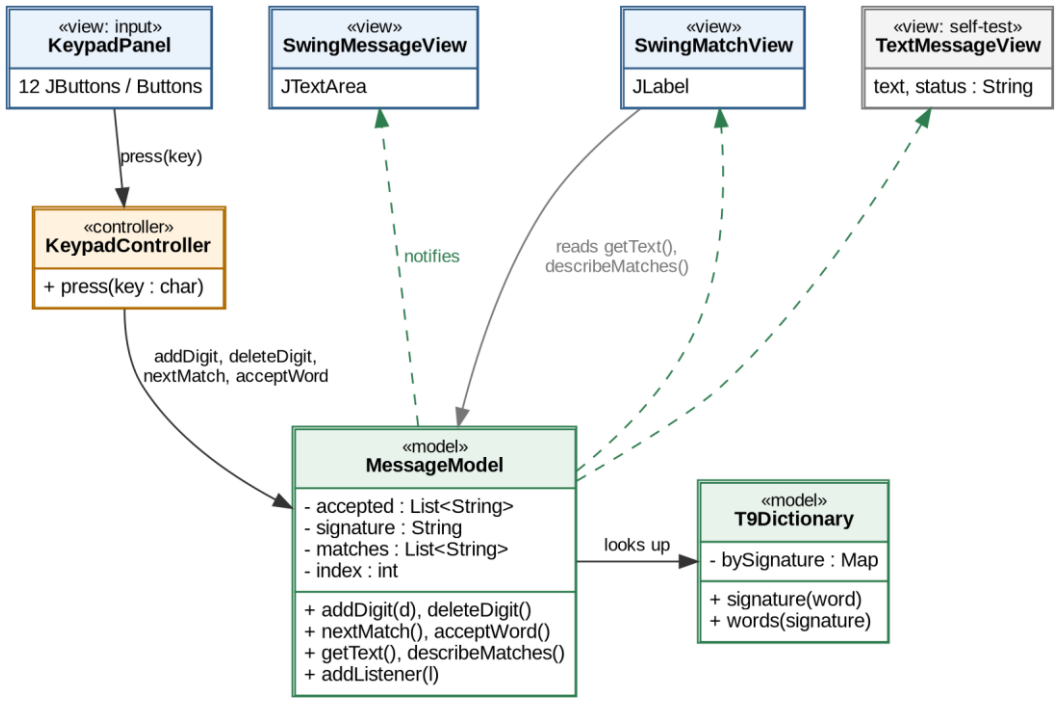


Figure 11.10 The classes of the predictive keypad. The keypad panel reports each press to the controller, which calls one of the model's four operations; the model uses the dictionary and notifies its three views, which read the text and the current match back from the model.

11.7.2 The program

Listing 11.7 is the Java program. It is the longest listing so far, but every class in it is short and has one job.

Listing 11.7 `PredictiveKeypad.java`: a phone keypad with predictive text. `T9Dictionary` and `MessageModel` form the model, `KeypadController` the controller, `SwingMessageView` and `SwingMatchView` two views of the same model, and `KeypadPanel` the twelve buttons; `TextMessageView` replaces the window in the self-test.

```
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.SwingUtilities;

public class PredictiveKeypad {
    static final List<String> BUILT_IN_WORDS = List.of(
            "a", "act", "and", "bat", "cat", "cod", "dry", "fold", "go", "gone",
            "good", "he", "hello", "hi", "home", "hone", "hood", "hoof", "i", "if",
            "in", "iron", "is", "it", "laundry", "mane", "me", "my", "name", "no",
            "oboe", "of", "on", "order", "ox", "ready", "shirt", "the", "tie", "to",
            "vie", "wash", "we", "world");

    public static void main(String[] args) throws IOException {
        boolean selftest = false;
        T9Dictionary dictionary = new T9Dictionary(BUILT_IN_WORDS);
        for (String arg : args) {
            if (arg.equals("--selftest")) {
                selftest = true;
            } else {
                dictionary = T9Dictionary.fromFile(Path.of(arg));  // e.g. a word list
            }
        }
        MessageModel model = new MessageModel(dictionary);
        KeypadController controller = new KeypadController(model);
        if (selftest) {
            selfTest(model, controller);
            return;
        }
        SwingUtilities.invokeLater(() -> {
            JFrame frame = new JFrame("Predictive Text");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            JPanel screen = new JPanel(new BorderLayout());
```

Listing 11.7 (Continued) *PredictiveKeypad.java: a phone keypad with predictive text. T9Dictionary and MessageModel form the model, KeypadController the controller, SwingMessageView and SwingMatchView two views of the same model, and KeypadPanel the twelve buttons; TextMessageView replaces the window in the self-test.*

```
            screen.add(new SwingMessageView(model), BorderLayout.CENTER);
            screen.add(new SwingMatchView(model), BorderLayout.SOUTH);
            frame.add(screen, BorderLayout.CENTER);
            frame.add(new KeypadPanel(controller), BorderLayout.SOUTH);
            frame.setSize(240, 320);
            frame.setLocationRelativeTo(null);
            frame.setVisible(true);
        });
    }

    // Press keys through the controller; a text view shows what the window would.
    static void selfTest(MessageModel model, KeypadController controller) {
        TextMessageView view = new TextMessageView(model);
        for (char key : "4355609675304663**092740143CC".toCharArray()) {
            controller.press(key);
            String line = String.format("key %c  %-26s %s",
                    key, "\"" + view.text() + "\"", view.status());
            System.out.println(line.stripTrailing());
        }
        if (!model.getText().equals("hello world home wash")) {
            throw new AssertionError("selftest failed: " + model.getText());
        }
        System.out.println("selftest passed: " + model.getWords().size() + " words");
    }
}

// Part of the model: which words each key sequence can mean.
class T9Dictionary {
    private static final String[] LETTERS =
            {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
    private final Map<String, List<String>> bySignature = new TreeMap<>();

    T9Dictionary(Collection<String> words) {
        for (String raw : words) {
            String word = raw.trim().toLowerCase();
            if (isValid(word)) {
                bySignature.computeIfAbsent(signature(word), k -> new
ArrayList<>()).add(word);
            }
        }
        for (List<String> list : bySignature.values()) {
            list.sort(null);  // alphabetical, so the order is predictable
        }
    }
```

Listing 11.7 (Continued) *PredictiveKeypad.java: a phone keypad with predictive text. T9Dictionary and MessageModel form the model, KeypadController the controller, SwingMessageView and SwingMatchView two views of the same model, and KeypadPanel the twelve buttons; TextMessageView replaces the window in the self-test.*

```
    static T9Dictionary fromFile(Path path) throws IOException {
        return new T9Dictionary(Files.readAllLines(path));
    }

    static boolean isValid(String word) {
        return !word.isEmpty() && word.chars().allMatch(c -> c >= 'a' && c <= 'z');
    }

    static String lettersOf(char key) {
        return LETTERS[key - '0'];
    }

    static String signature(String word) {
        StringBuilder sig = new StringBuilder();
        for (char c : word.toCharArray()) {
            for (int key = 2; key <= 9; key++) {
                if (LETTERS[key].indexOf(c) >= 0) {
                    sig.append((char) ('0' + key));
                }
            }
        }
        return sig.toString();
    }

    List<String> words(String signature) {
        return List.copyOf(bySignature.getOrDefault(signature, List.of()));
    }
}

// Model: the message being typed. It knows nothing about buttons or windows.
class MessageModel {
    private final T9Dictionary dictionary;
    private final List<String> accepted = new ArrayList<>();
    private final List<Runnable> listeners = new ArrayList<>();
    private String signature = "";
    private List<String> matches = List.of();
    private int index;

    MessageModel(T9Dictionary dictionary) {
        this.dictionary = dictionary;
    }

    void addListener(Runnable listener) {
        listeners.add(listener);
    }
```

Listing 11.7 (Continued) *PredictiveKeypad.java: a phone keypad with predictive text. T9Dictionary and MessageModel form the model, KeypadController the controller, SwingMessageView and SwingMatchView two views of the same model, and KeypadPanel the twelve buttons; TextMessageView replaces the window in the self-test.*

```
    private void fireChanged() {
        for (Runnable listener : listeners) {
            listener.run();
        }
    }

    void addDigit(char digit) {
        if (digit < '2' || digit > '9') {
            throw new IllegalArgumentException("not a letter key: " + digit);
        }
        signature += digit;
        lookUp();
        fireChanged();
    }

    void deleteDigit() {
        if (!signature.isEmpty()) {
            signature = signature.substring(0, signature.length() - 1);
            lookUp();
            fireChanged();
        }
    }

    void nextMatch() {
        if (matches.size() > 1) {
            index = (index + 1) % matches.size();  // wrap around to the first
            fireChanged();
        }
    }

    void acceptWord() {
        if (!signature.isEmpty()) {
            accepted.add(currentWord());
            signature = "";
            lookUp();
            fireChanged();
        }
    }

    private void lookUp() {
        matches = dictionary.words(signature);
        index = 0;
    }
```

Listing 11.7 (Continued) *PredictiveKeypad.java: a phone keypad with predictive text. T9Dictionary and MessageModel form the model, KeypadController the controller, SwingMessageView and SwingMatchView two views of the same model, and KeypadPanel the twelve buttons; TextMessageView replaces the window in the self-test.*

```
    String getSignature() {
        return signature;
    }

    List<String> getMatches() {
        return matches;
    }

    int getMatchIndex() {
        return index;
    }

    // An unknown key sequence is shown as its digits.
    String currentWord() {
        return matches.isEmpty() ? signature : matches.get(index);
    }

    List<String> getWords() {
        List<String> all = new ArrayList<>(accepted);
        if (!signature.isEmpty()) {
            all.add(currentWord());
        }
        return all;
    }

    String getText() {
        return String.join(" ", getWords());
    }

    String describeMatches() {
        if (signature.isEmpty()) {
            return "";
        }
        if (matches.isEmpty()) {
            return signature + " -> no word";
        }
        return signature + " -> " + currentWord()
                + " (" + (index + 1) + " of " + matches.size() + ")";
    }
}

// Controller: translates key presses into model operations.
class KeypadController {
    private final MessageModel model;
```

Listing 11.7 (Continued) *PredictiveKeypad.java: a phone keypad with predictive text. T9Dictionary and MessageModel form the model, KeypadController the controller, SwingMessageView and SwingMatchView two views of the same model, and KeypadPanel the twelve buttons; TextMessageView replaces the window in the self-test.*

```
    KeypadController(MessageModel model) {
        this.model = model;
    }

    void press(char key) {
        switch (key) {
            case '0' -> model.acceptWord();
            case '*' -> model.nextMatch();
            case 'C' -> model.deleteDigit();
            case '1' -> { }  // key 1 carries no letters
            default -> model.addDigit(key);
        }
    }
}

// View 1: the message itself, in a read-only text area.
class SwingMessageView extends JPanel {
    private final JTextArea message = new JTextArea(5, 20);

    SwingMessageView(MessageModel model) {
        super(new BorderLayout());
        message.setEditable(false);
        message.setLineWrap(true);
        message.setWrapStyleWord(true);
        message.setText("predictive text: enter text with 8 keys.");
        add(new JScrollPane(message), BorderLayout.CENTER);
        model.addListener(() -> message.setText(model.getText()));
    }
}

// View 2: the key sequence and the candidate being shown.
class SwingMatchView extends JLabel {
    SwingMatchView(MessageModel model) {
        super(" ");
        model.addListener(() -> setText(" " + model.describeMatches()));
    }
}
```

Listing 11.7 (Continued) *PredictiveKeypad.java: a phone keypad with predictive text. T9Dictionary and MessageModel form the model, KeypadController the controller, SwingMessageView and SwingMatchView two views of the same model, and KeypadPanel the twelve buttons; TextMessageView replaces the window in the self-test.*

```
// The keypad: twelve buttons that report presses to the controller.
class KeypadPanel extends JPanel {
    KeypadPanel(KeypadController controller) {
        super(new GridLayout(4, 3));
        for (char key : "123456789*0C".toCharArray()) {
            String letters = switch (key) {
                case '0' -> "_";
                case '*', 'C', '1' -> "";
                default -> T9Dictionary.lettersOf(key);
            };
            JButton button = new JButton(
                    "<html><center>" + key + "<br>" + letters + "</center></html>");
            button.addActionListener(e -> controller.press(key));
            add(button);
        }
    }
}

// A stand-in view for the self-test: it renders into strings.
class TextMessageView {
    private String text = "predictive text: enter text with 8 keys.";
    private String status = "";

    TextMessageView(MessageModel model) {
        model.addListener(() -> {
            text = model.getText();
            status = model.describeMatches();
        });
    }

    String text() {
        return text;
    }

    String status() {
        return status;
    }
}
```

Let us read it class by class, from the bottom of the architecture upward.

`T9Dictionary` is part of the model. The array `LETTERS` holds, at index k, the letters of key k (keys 0 and 1 have none). `signature(word)` is Equation (11.3) in code. The original `wordToSignature` used an eight-branch `if`–`else` chain; a table is shorter and, more importantly, it is the *same* table that labels the keypad's buttons, so labels and lookup cannot disagree. (They did disagree in the original: key 9's button was labeled *vwxy*, while the lookup correctly used *wxyz*. Exercise 11.6 asks why no test of the model could have caught that.) The constructor builds the map from signature to words with `computeIfAbsent`, which creates an empty list the first time a signature is seen, and then sorts every list alphabetically so that the order of matches is predictable. The original used a `HashSet`, whose order depends on hash codes. `isValid` keeps only words made of the letters *a* to *z*. And `fromFile` reads one word per line from any file, such as `/usr/share/dict/words` on many Linux and macOS systems; the original program *required* that file and could not run without it, while ours carries a small built-in list of 44 words and uses a file only if you name one on the command line.

`MessageModel` is the heart of the model. It holds the accepted words, the current signature, the current matches and the index of the match shown, and a list of listeners of type `Runnable` — the pull style of Section 11.5.4: a listener is told only *that* something changed and asks the model for what it needs. Each of the four operations of **Table 11.3** checks whether it has anything to do, changes the state, and calls `fireChanged()`. The checks matter. `deleteDigit` on an empty signature, `nextMatch` with fewer than two matches and `acceptWord` with nothing typed change nothing, and so they fire nothing: Equation (11.2) again. The original `press` method notified after every key, even when nothing had changed, and its `'0'` added an empty word to the message when pressed twice, producing double spaces. `addDigit` rejects anything but `'2'` to `'9'` with an `IllegalArgumentException`: the model protects its own invariant, whatever the controller does. The query methods — `currentWord`, `getWords`, `getText` and `describeMatches` — compute what the views need from the state; `currentWord` shows the digits when there is no match.

`KeypadController` is short: `press(key)` is the table **Table 11.3** written as a `switch`. Its job is translation. It knows that the button labeled `0` means "accept the word" on this keypad, which is a fact about the *user interface*, not about messages — on a different keyboard, the space bar would mean the same thing. That is exactly the knowledge that belongs in a controller.

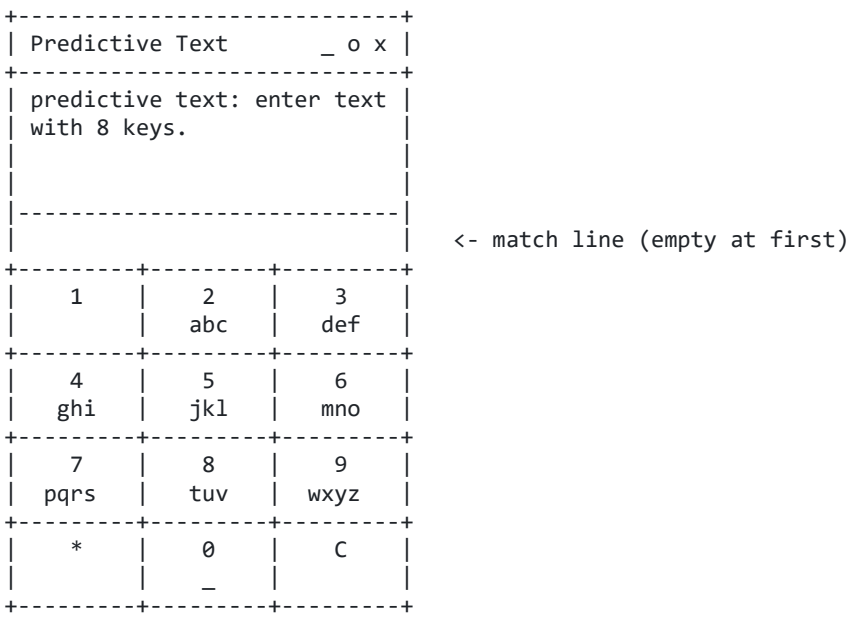
`SwingMessageView` and `SwingMatchView` are two views of the same model. The first is a `JPanel` holding a read-only `JTextArea` of 5 rows and 20 columns, with line wrapping at word boundaries, inside a `JScrollPane`; it subscribes to the model with a lambda that copies `model.getText()` into the text area. The second is a `JLabel` that shows `describeMatches()`, for example `4663 -> home (3 of 6)`, so the user can see which of several candidates is displayed. Neither view knows about the other; both are kept current by the same notifications — the situation of **Figure 11.4** in miniature.

`KeypadPanel` builds the twelve buttons in a `GridLayout` of four rows and three columns, in the order of a phone: 1 2 3, 4 5 6, 7 8 9, `*` 0 C. Each button's text is a snippet of HTML — `<html><center>2<br>abc</center></html>` — because a Swing button accepts simple HTML, which is the easiest way to put the letters on a second line. The letters come from `T9Dictionary.lettersOf`. Each button's listener calls `controller.press(key)`: the panel is the input half of the view, and it forwards gestures without interpreting them.

`main` builds the dictionary, the model and the controller, and then either runs the self-test or builds the window on the event dispatch thread.

11.7.3 The window

Run `java PredictiveKeypad`. A window titled **Predictive Text**, 240 pixels wide and 320 high, appears in the middle of the screen:



The top part is the message area, white, read-only, showing the greeting *predictive text: enter text with 8 keys.* until the first key is pressed. Below it runs a one-line label for the current match, empty at first. The lower part is the keypad: twelve buttons of equal size in four rows, each showing its digit and, below it, its letters; key 0 shows an underscore as a picture of a space. Pressing 4, 3, 5, 5, 6 makes the message area show *i*, *he*, *435*, *4355* and finally *hello*, while the match line shows, for instance, `43 -> he (1 of 2)` and then `43556 -> hello (1 of 1)`. Pressing 0 accepts the word; pressing 4, 6, 6, 3 shows *gone*, and pressing `*` twice changes it to *good* and then *home*, with the match line reading `4663 -> home (3 of 6)`.

11.7.4 The self-test

`java PredictiveKeypad --selftest` presses the keys of a whole message through the controller — exactly the calls the buttons would make — while a `TextMessageView` records what the two window views would show. After each key the self-test prints the key, the message text and the match line:

```
key 4  "i"                        4 -> i (1 of 1)
key 3  "he"                       43 -> he (1 of 2)
key 5  "435"                      435 -> no word
key 5  "4355"                     4355 -> no word
key 6  "hello"                    43556 -> hello (1 of 1)
key 0  "hello"
key 9  "hello 9"                  9 -> no word
key 6  "hello 96"                 96 -> no word
key 7  "hello 967"                967 -> no word
key 5  "hello 9675"               9675 -> no word
key 3  "hello world"              96753 -> world (1 of 1)
key 0  "hello world"
key 4  "hello world i"            4 -> i (1 of 1)
key 6  "hello world go"           46 -> go (1 of 2)
key 6  "hello world 466"          466 -> no word
key 3  "hello world gone"         4663 -> gone (1 of 6)
key *  "hello world good"         4663 -> good (2 of 6)
key *  "hello world home"         4663 -> home (3 of 6)
key 0  "hello world home"
key 9  "hello world home 9"       9 -> no word
key 2  "hello world home 92"      92 -> no word
key 7  "hello world home 927"     927 -> no word
key 4  "hello world home wash"    9274 -> wash (1 of 1)
key 0  "hello world home wash"
key 1  "hello world home wash"
key 4  "hello world home wash i"  4 -> i (1 of 1)
key 3  "hello world home wash he" 43 -> he (1 of 2)
key C  "hello world home wash i"  4 -> i (1 of 1)
key C  "hello world home wash"
selftest passed: 4 words
```

Read the output as a film of the screen. The first five keys spell *hello*, and the intermediate states show the dictionary at work: after `4` the only match is the one-letter word *i*; after `43` there are two matches, *he* and *if*, and the first in alphabetical order is shown; `435` and `4355` match nothing, so the digits are displayed; `43556` is *hello*. Key `0` accepts the word, and the match line empties because a new word has begun. *world* follows the same pattern through five keys. Then `4663` produces *gone*, the first of six matches, and two presses of `*` move to *good* and *home*. After `0` and `9274` — *wash* — and another `0`, the key `1` changes nothing, as **Table 11.3** promises. The last four keys show deletion: `4` and `3` begin *he*, and two presses of `C` remove the two digits again, leaving the message *hello world home wash*. The self-test checks that final text and prints its verdict.

The self-test found a real bug while this chapter was being written: the first version of its key string had `466` where `4663` was meant, and the closing check failed with `selftest failed: hello world 466 wash`. It cannot, however, see that a button label is misspelled; it checks the program's *logic*, in a second, after every change.

11.7.5 The same keypad in Python

Listing 11.8 is the Python version. The model and the controller are line-for-line translations; the window is built by one class, `TkKeypadWindow`, which holds both views and the keypad.

Listing 11.8 `predictive_keypad.py`: the predictive keypad in Python. The model and controller match the Java classes; `TkKeypadWindow` builds the message box, the match line and the twelve buttons with Tkinter.

```
"""predictive_keypad.py: a phone keypad with predictive text, as MVC."""
import sys

BUILT_IN_WORDS = [
    "a", "act", "and", "bat", "cat", "cod", "dry", "fold", "go", "gone",
    "good", "he", "hello", "hi", "home", "hone", "hood", "hoof", "i", "if",
    "in", "iron", "is", "it", "laundry", "mane", "me", "my", "name", "no",
    "oboe", "of", "on", "order", "ox", "ready", "shirt", "the", "tie", "to",
    "vie", "wash", "we", "world"]

LETTERS = ["", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"]


class T9Dictionary:
    """Part of the model: which words each key sequence can mean."""

    def __init__(self, words):
        self._by_signature = {}
        for raw in words:
            word = raw.strip().lower()
            if self.is_valid(word):
                self._by_signature.setdefault(self.signature(word), []).append(word)
        for matches in self._by_signature.values():
            matches.sort()  # alphabetical, so the order is predictable

    @classmethod
    def from_file(cls, path):
        with open(path, encoding="utf-8") as f:
            return cls(f)

    @staticmethod
    def is_valid(word):
        return word != "" and all("a" <= c <= "z" for c in word)

    @staticmethod
    def letters_of(key):
        return LETTERS[int(key)]

    @staticmethod
    def signature(word):
        return "".join(str(key) for c in word
                       for key in range(2, 10) if c in LETTERS[key])

    def words(self, signature):
        return list(self._by_signature.get(signature, []))
```

Listing 11.8 (Continued) *predictive_keypad.py: the predictive keypad in Python. The model and controller match the Java classes; TkKeypadWindow builds the message box, the match line and the twelve buttons with Tkinter.*

```
class MessageModel:
    """Model: the message being typed. It knows nothing about buttons or windows."""

    def __init__(self, dictionary):
        self._dictionary = dictionary
        self._accepted = []
        self._listeners = []
        self.signature = ""
        self.matches = []
        self.index = 0

    def add_listener(self, listener):
        self._listeners.append(listener)

    def _fire_changed(self):
        for listener in list(self._listeners):
            listener()

    def add_digit(self, digit):
        if len(digit) != 1 or digit not in "23456789":
            raise ValueError("not a letter key: " + digit)
        self.signature += digit
        self._look_up()
        self._fire_changed()

    def delete_digit(self):
        if self.signature:
            self.signature = self.signature[:-1]
            self._look_up()
            self._fire_changed()

    def next_match(self):
        if len(self.matches) > 1:
            self.index = (self.index + 1) % len(self.matches)  # wrap around
            self._fire_changed()

    def accept_word(self):
        if self.signature:
            self._accepted.append(self.current_word())
            self.signature = ""
            self._look_up()
            self._fire_changed()
```

Listing 11.8 (Continued) *predictive_keypad.py: the predictive keypad in Python. The model and controller match the Java classes; TkKeypadWindow builds the message box, the match line and the twelve buttons with Tkinter.*

```
    def _look_up(self):
        self.matches = self._dictionary.words(self.signature)
        self.index = 0

    def current_word(self):
        # An unknown key sequence is shown as its digits.
        return self.matches[self.index] if self.matches else self.signature

    def words(self):
        return self._accepted + ([self.current_word()] if self.signature else [])

    def text(self):
        return " ".join(self.words())

    def describe_matches(self):
        if not self.signature:
            return ""
        if not self.matches:
            return self.signature + " -> no word"
        return (f"{self.signature} -> {self.current_word()} "
                f"({self.index + 1} of {len(self.matches)})")


class KeypadController:
    """Controller: translates key presses into model operations."""

    def __init__(self, model):
        self._model = model

    def press(self, key):
        if key == "0":
            self._model.accept_word()
        elif key == "*":
            self._model.next_match()
        elif key == "C":
            self._model.delete_digit()
        elif key == "1":
            pass  # key 1 carries no letters
        else:
            self._model.add_digit(key)


class TkKeypadWindow:
    """Both views and the keypad, in one Tkinter window."""
```

Listing 11.8 (Continued) *predictive_keypad.py: the predictive keypad in Python. The model and controller match the Java classes; TkKeypadWindow builds the message box, the match line and the twelve buttons with Tkinter.*

```
    def __init__(self, model, controller):
        import tkinter as tk  # imported here, so the self-test needs no Tkinter
        self._model = model
        self._root = tk.Tk()
        self._root.title("Predictive Text")
        self._root.geometry("240x320")
        # View 1: the message itself, in a read-only text box.
        self._message = tk.Text(self._root, height=5, width=20, wrap="word")
        self._message.insert("1.0", "predictive text: enter text with 8 keys.")
        self._message.config(state="disabled")
        self._message.pack(fill="both", expand=True)
        # View 2: the key sequence and the candidate being shown.
        self._status = tk.Label(self._root, text=" ", anchor="w")
        self._status.pack(fill="x")
        # The keypad: twelve buttons that report presses to the controller.
        pad = tk.Frame(self._root)
        pad.pack(fill="x")
        for i, key in enumerate("123456789*0C"):
            letters = ("_" if key == "0" else "" if key in "*C1"
                       else T9Dictionary.letters_of(key))
            tk.Button(pad, text=f"{key}\n{letters}", width=6,
                      command=lambda k=key: controller.press(k)).grid(
                row=i // 3, column=i % 3, sticky="nsew")
        for column in range(3):
            pad.columnconfigure(column, weight=1)
        model.add_listener(self.refresh)

    def refresh(self):
        self._message.config(state="normal")
        self._message.delete("1.0", "end")
        self._message.insert("1.0", self._model.text())
        self._message.config(state="disabled")
        self._status.config(text=" " + self._model.describe_matches())

    def open(self):
        self._root.mainloop()


class TextMessageView:
    """A stand-in view for the self-test: it renders into strings."""

    def __init__(self, model):
        self.text = "predictive text: enter text with 8 keys."
        self.status = ""
```

Listing 11.8 (Continued) *predictive_keypad.py: the predictive keypad in Python. The model and controller match the Java classes; TkKeypadWindow builds the message box, the match line and the twelve buttons with Tkinter.*

```
        def refresh():
            self.text = model.text()
            self.status = model.describe_matches()
        model.add_listener(refresh)


def self_test(model, controller):
    """Press keys through the controller; a text view shows what the window would."""
    view = TextMessageView(model)
    for key in "4355609675304663**092740143CC":
        controller.press(key)
        quoted = '"' + view.text + '"'
        line = f"key {key}  {quoted:<26} {view.status}"
        print(line.rstrip())
    assert model.text() == "hello world home wash", "selftest failed: " + model.text()
    print(f"selftest passed: {len(model.words())} words")


def main(args):
    dictionary = T9Dictionary(BUILT_IN_WORDS)
    for arg in args:
        if arg != "--selftest":
            dictionary = T9Dictionary.from_file(arg)  # e.g. a word list
    model = MessageModel(dictionary)
    controller = KeypadController(model)
    if "--selftest" in args:
        self_test(model, controller)
        return
    TkKeypadWindow(model, controller).open()


if __name__ == "__main__":
    main(sys.argv[1:])
```

`python3 predictive_keypad.py --selftest` produces output identical to Java's, character for character:

```
key 4  "i"                        4 -> i (1 of 1)
key 3  "he"                       43 -> he (1 of 2)
key 5  "435"                      435 -> no word
key 5  "4355"                     4355 -> no word
key 6  "hello"                    43556 -> hello (1 of 1)
key 0  "hello"
key 9  "hello 9"                  9 -> no word
key 6  "hello 96"                 96 -> no word
key 7  "hello 967"                967 -> no word
key 5  "hello 9675"               9675 -> no word
key 3  "hello world"              96753 -> world (1 of 1)
key 0  "hello world"
key 4  "hello world i"            4 -> i (1 of 1)
key 6  "hello world go"           46 -> go (1 of 2)
key 6  "hello world 466"          466 -> no word
key 3  "hello world gone"         4663 -> gone (1 of 6)
key *  "hello world good"         4663 -> good (2 of 6)
key *  "hello world home"         4663 -> home (3 of 6)
key 0  "hello world home"
key 9  "hello world home 9"       9 -> no word
key 2  "hello world home 92"      92 -> no word
key 7  "hello world home 927"     927 -> no word
key 4  "hello world home wash"    9274 -> wash (1 of 1)
key 0  "hello world home wash"
key 1  "hello world home wash"
key 4  "hello world home wash i"  4 -> i (1 of 1)
key 3  "hello world home wash he" 43 -> he (1 of 2)
key C  "hello world home wash i"  4 -> i (1 of 1)
key C  "hello world home wash"
selftest passed: 4 words
```

A few Python details are worth a sentence each. `signature` is a single generator expression: for each character `c` of the word, and for each key from 2 to 9, it produces the key's digit if `c` is among that key's letters. `setdefault(key, [])` plays the role of Java's `computeIfAbsent`. `from_file` passes the open file object straight to the constructor, because iterating over a file yields its lines. In the Tkinter window, the message is a `Text` widget of 5 lines and 20 characters with `wrap="word"`; a `Text` widget can only be changed while its state is `"normal"`, so `refresh` switches it to normal, replaces the contents, and switches it back to `"disabled"` so that the user cannot type into it. The buttons are created in a loop, and each one's command is `lambda k=key: controller.press(k)`. The `k=key` is essential: without it, all twelve lambdas would refer to the *same* variable `key`, which after the loop holds `"C"`, and every button would delete a digit. The default argument captures the value at the moment each lambda is created — the Python counterpart of Java's rule that a lambda may only capture effectively final variables, which Chapter 10 discussed.

11.8 Testing the Model Without the GUI

The self-tests of the last two sections run a *scenario*: a sequence of clicks, printed as it happens, with one check at the end. That is valuable, but it is not a unit test. A **unit test** checks one small behavior in isolation, has a descriptive name, and reports clearly which behavior failed. Chapter 6 introduced unit tests with JUnit 5 and Python's `unittest`. Because our keypad model imports nothing from Swing or Tkinter, we can test it exactly like any other class, with no window, no event thread and no screen.

What should the tests check? A good rule is to test each *promise* the model makes, including the promises about what it will *not* do:

- the signature function computes Equation (11.3) correctly;

- invalid words are rejected by the dictionary;
- matches come back in alphabetical order;
- an unknown key sequence is shown as its digits;
- `nextMatch` wraps round after the last match;
- the model fires exactly one event per real change and none for an operation that changes nothing;
- the model refuses a digit that is not a letter key;
- the controller maps keys to the right operations.

Listing 11.9 tests these promises in plain Java: a small `check` method compares an expected and an actual value and prints `PASS` or `FAIL`, in the style of the check programs of Chapter 6. With JUnit each block would become a method annotated with `@Test` and each `check` an `assertEquals` (the Jupiter API of JUnit 5 is unchanged in JUnit 6, the current major version, 6.1.3 in August 2026, which needs Java 17 or later [12]); the plain version runs anywhere with nothing but the JDK, which is why the book uses it here.

Listing 11.9 `MessageModelCheck.java`: ten checks of the keypad model and controller, run without any window.

```
import java.util.List;
import java.util.Objects;

public class MessageModelCheck {
    private static int passed;
    private static int failed;

    public static void main(String[] args) {
        check("signature of hello", "43556", T9Dictionary.signature("hello"));
        check("signature of wash", "9274", T9Dictionary.signature("wash"));
        check("words are rejected unless a-z", false, T9Dictionary.isValid("don't"));

        MessageModel model = newModel();
        type(model, "4663");
        check("matches are alphabetical",
                List.of("gone", "good", "home", "hone", "hood", "hoof"),
model.getMatches());

        model = newModel();
        type(model, "96");
        check("an unknown sequence shows its digits", "96", model.getText());

        model = newModel();
        type(model, "4663");
        for (int i = 0; i < 6; i++) {
            model.nextMatch();
        }
        check("six presses of * wrap round to the first match", "gone",
model.currentWord());

        model = newModel();
        int[] events = {0};
        model.addListener(() -> events[0]++);
        type(model, "43556");
        model.acceptWord();
        model.acceptWord();  // nothing to accept: no change, no event
        model.deleteDigit(); // nothing to delete: no change, no event
        check("one event per real change", 6, events[0]);
        check("accepted text", "hello", model.getText());

        try {
            newModel().addDigit('1');
            check("key 1 is refused", "exception", "no exception");
        } catch (IllegalArgumentException e) {
            check("key 1 is refused", "not a letter key: 1", e.getMessage());
        }
```

Listing 11.9 (Continued) *MessageModelCheck.java: ten checks of the keypad model and controller, run without any window.*

```
        model = newModel();
        KeypadController controller = new KeypadController(model);
        for (char key : "8430*0".toCharArray()) {
            controller.press(key);
        }
        check("controller: 843, accept, then * and 0 do nothing", "the", model.getText());

        System.out.println(passed + " of " + (passed + failed) + " checks passed");
        if (failed > 0) {
            System.exit(1);
        }
    }

    static MessageModel newModel() {
        return new MessageModel(new T9Dictionary(PredictiveKeypad.BUILT_IN_WORDS));
    }

    static void type(MessageModel model, String digits) {
        for (char d : digits.toCharArray()) {
            model.addDigit(d);
        }
    }

    static void check(String name, Object expected, Object actual) {
        if (Objects.equals(expected, actual)) {
            passed++;
            System.out.println("PASS " + name);
        } else {
            failed++;
            System.out.println("FAIL " + name + ": expected " + expected + ", got " +
actual);
        }
    }
}
```

Compile it together with the keypad (`javac *.java` in the chapter's `java` folder compiles everything) and run `java MessageModelCheck`:

```
PASS signature of hello
PASS signature of wash
PASS words are rejected unless a-z
PASS matches are alphabetical
PASS an unknown sequence shows its digits
PASS six presses of * wrap round to the first match
PASS one event per real change
PASS accepted text
PASS key 1 is refused
PASS controller: 843, accept, then * and 0 do nothing
10 of 10 checks passed
```

The most interesting check is *one event per real change*: a counting listener sees five digits and one accepted word, and nothing for the `acceptWord` and `deleteDigit` calls that have nothing to do — 6 events, not 8. (The counter is a one-element array because a Java lambda may not assign to a local variable of the enclosing method; it may, however, change the contents of an array that the variable refers to.) The *key 1 is refused* check shows how to test an exception in plain Java: call the method inside `try`, report a failure if it returns normally, and compare the exception's message in `catch`. The last check runs the controller and the model together: `8430*0` types *the*, accepts it, and then presses `*` and `0` with nothing typed, which must change nothing.

The program ends with `System.exit(1)` if any check failed; a non-zero exit status tells the operating system, and any automatic build, that it failed.

Listing 11.10 is the same set of tests with Python's `unittest`.

Listing 11.10 `test_keypad_model.py`: unit tests of the keypad model and controller with `unittest`.

```
"""test_keypad_model.py: unit tests for the keypad model, with no window at all."""
import unittest

from predictive_keypad import (BUILT_IN_WORDS, KeypadController, MessageModel,
                               T9Dictionary)


def new_model():
    return MessageModel(T9Dictionary(BUILT_IN_WORDS))


def type_digits(model, digits):
    for d in digits:
        model.add_digit(d)


class DictionaryTest(unittest.TestCase):
    def test_signatures(self):
        self.assertEqual(T9Dictionary.signature("hello"), "43556")
        self.assertEqual(T9Dictionary.signature("wash"), "9274")

    def test_only_a_to_z_words_are_valid(self):
        self.assertFalse(T9Dictionary.is_valid("don't"))


class MessageModelTest(unittest.TestCase):
    def setUp(self):
        self.model = new_model()  # a fresh model before every test
        self.events = 0

        def count():
            self.events += 1
        self.model.add_listener(count)

    def test_matches_are_alphabetical(self):
        type_digits(self.model, "4663")
        self.assertEqual(self.model.matches,
                         ["gone", "good", "home", "hone", "hood", "hoof"])

    def test_unknown_sequence_shows_its_digits(self):
        type_digits(self.model, "96")
        self.assertEqual(self.model.text(), "96")
```

Listing 11.10 (Continued) *test_keypad_model.py: unit tests of the keypad model and controller with unittest.*

```
    def test_star_wraps_round(self):
        type_digits(self.model, "4663")
        for _ in range(6):
            self.model.next_match()
        self.assertEqual(self.model.current_word(), "gone")

    def test_one_event_per_real_change(self):
        type_digits(self.model, "43556")
        self.model.accept_word()
        self.model.accept_word()   # nothing to accept: no change, no event
        self.model.delete_digit()  # nothing to delete: no change, no event
        self.assertEqual(self.events, 6)
        self.assertEqual(self.model.text(), "hello")

    def test_key_1_is_refused(self):
        with self.assertRaises(ValueError) as caught:
            self.model.add_digit("1")
        self.assertEqual(str(caught.exception), "not a letter key: 1")


class KeypadControllerTest(unittest.TestCase):
    def test_controller_maps_keys_to_operations(self):
        model = new_model()
        controller = KeypadController(model)
        for key in "8430*0":
            controller.press(key)
        self.assertEqual(model.text(), "the")


if __name__ == "__main__":
    unittest.main()
```

Run the tests with `python3 -m unittest -v test_keypad_model` in the chapter's `python` folder (the `-v` asks for one line per test):

```
test_only_a_to_z_words_are_valid
(test_keypad_model.DictionaryTest.test_only_a_to_z_words_are_valid) ... ok
test_signatures (test_keypad_model.DictionaryTest.test_signatures) ... ok
test_controller_maps_keys_to_operations
(test_keypad_model.KeypadControllerTest.test_controller_maps_keys_to_operations) ... ok
test_key_1_is_refused (test_keypad_model.MessageModelTest.test_key_1_is_refused) ... ok
test_matches_are_alphabetical
(test_keypad_model.MessageModelTest.test_matches_are_alphabetical) ... ok
test_one_event_per_real_change
(test_keypad_model.MessageModelTest.test_one_event_per_real_change) ... ok
test_star_wraps_round (test_keypad_model.MessageModelTest.test_star_wraps_round) ... ok
test_unknown_sequence_shows_its_digits
(test_keypad_model.MessageModelTest.test_unknown_sequence_shows_its_digits) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.001s

OK
```

`unittest` finds every method whose name starts with `test` in every subclass of `TestCase`, runs each one separately, and reports `ok` or the reason for failure. The method `setUp` runs before *each* test, so every test of `MessageModelTest` starts with a fresh model and an event counter at zero; tests that share state can make each other fail in confusing ways, and a fresh model for each test prevents that. `assertRaises` used as a context manager (`with self.assertRaises(ValueError) as caught:`) checks that the block raises the exception, and `caught.exception` lets us check its message as well. The tests run in alphabetical order of their names; well-written tests do not

depend on their order. They took a few milliseconds — fast enough to run after every single change, which a test that needs a window never is.

> ✓ **KEY INSIGHT.** The question "how do I test my GUI?" is usually the wrong question. Ask instead "how little logic can I leave in my GUI?" Move every decision into the model or the controller, where a unit test can reach it in milliseconds, and keep the view so thin that it has nothing left to test but its appearance.

11.9 Profitina Laundry Front Desk: A Complete MVC Application

We now return to the story of Section 11.2 and build the front-desk program properly. The requirements are those the clerk already knows:

1. The clerk enters a customer name, chooses one of three services, types the weight in kilograms and may tick *Express*; clicking **Add order** records the order with the next order number and shows its price.
2. Invalid input is refused with a clear message and changes nothing: an empty name, a weight that is not a number, a weight of zero or less, or more than 50 kg (the capacity of the laundry's largest machine). Because Indonesians write decimal numbers with a comma, `2,6` must be accepted as 2.6.
3. The orders are listed with their number, customer, service, weight, express flag, price and status.
4. The clerk selects an order and clicks **Advance status** to move it one step forward: *Received*, *Washing*, *Ready*, *Picked up*. An order that has been picked up cannot advance further.
5. A summary line shows the number of orders, how many are still *Received*, how many are *Ready* for pickup, and the total revenue.

11.9.1 The rules of the model

The prices follow the rules of Chapter 7. Each service s has a rate r_s in rupiah per kilogram and a minimum weight w_s: Wash & Fold has $r = 7000$ and $w = 3$ kg, Wash, Dry & Iron has $r = 10000$ and $w = 3$ kg, and Iron Only has $r = 5000$ and $w = 2$ kg. A customer who brings w kilograms is charged for at least the minimum, rounded *up* to the next half kilogram. Rounding a number x up to the next half is done by doubling it, rounding up to a whole number with the ceiling function $\lceil \cdot \rceil$, and halving again, so the **chargeable weight** is

$$\hat{w} = \frac{\lceil 2 \max(w, w_s) \rceil}{2} \tag{11.4}$$

and the **base price** is

$$p = r_s \cdot \hat{w} \tag{11.5}$$

An express order costs half as much again:

$$p_{\text{ex}} = \left\lfloor \frac{3p}{2} \right\rfloor \tag{11.6}$$

where $\lfloor \cdot \rfloor$ rounds down; since every base price here is a multiple of 500, $3p/2$ is always a whole number and the rounding never actually happens, but the program uses integer arithmetic and the formula says exactly what it computes.

Let us work the three orders of the self-test by hand, one step per line. Sari brings 4.2 kg of Wash & Fold. By Equation (11.4), $\max(4.2,3) = 4.2$; $2 \times 4.2 = 8.4$; $\lceil 8.4 \rceil = 9$; $9/2 = 4.5$ kg. By Equation (11.5), $p = 7000 \times 4.5 = 31500$, so Rp31,500. Budi brings 2.6 kg of Wash, Dry & Iron, express. $\max(2.6,3) = 3$; $\lceil 6 \rceil / 2 = 3$ kg; $p = 10000 \times 3 = 30000$; by Equation (11.6), $p_{\text{ex}} = \lfloor 90000/2 \rfloor = 45000$, so Rp45,000. Citra brings 1.3 kg to be ironed: $\max(1.3,2) = 2$ kg, $p = 5000 \times 2 = 10000$. The total is $31500 + 45000 + 10000 = 86500$, Rp86,500 — the number the self-test and the unit tests will check.

The statuses form a small **state machine**, drawn in **Figure 11.11**. An order is born *Received* by `addOrder`; each `advance()` moves it one step to the right; in the state *Picked up*, `advance()` is refused with an exception and the order stays where it is.

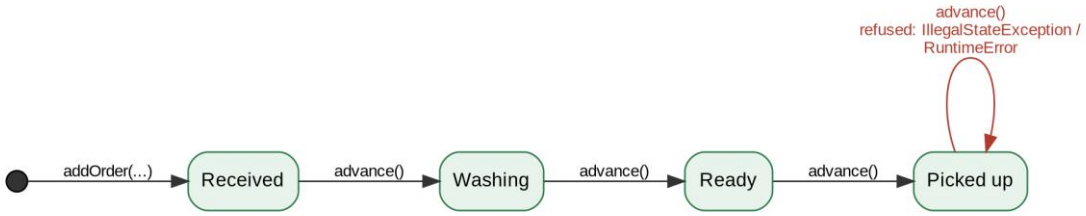


Figure 11.11 The life of a laundry order. Each `advance()` moves an order one state to the right; an order that has been picked up refuses to advance.

11.9.2 The design

Figure 11.12 shows every class of the application. It follows the mediator arrangement of **Figure 11.5**: the model notifies the controller, and the controller updates the views, so the views never hold a reference to the model.

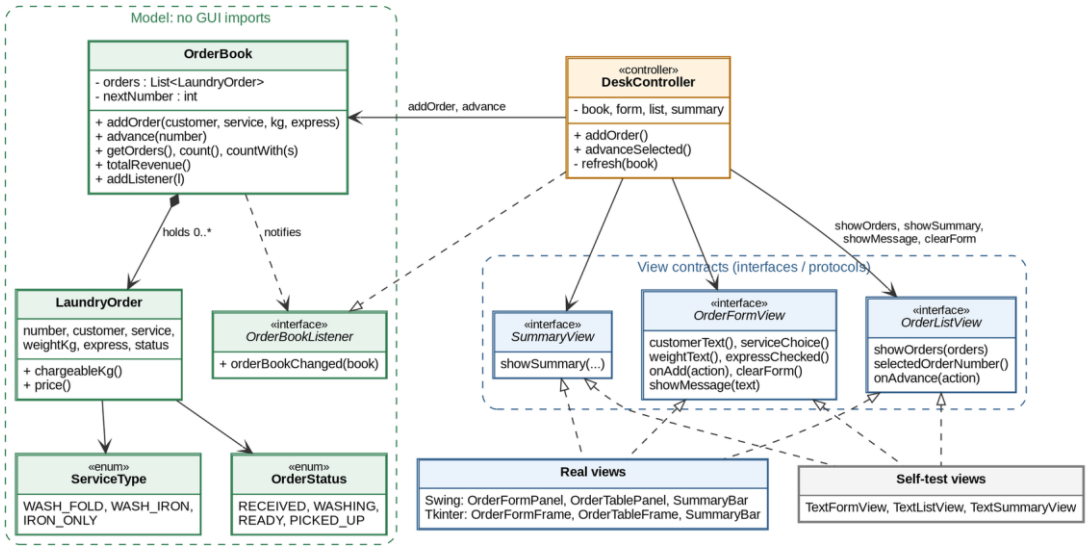


Figure 11.12 Class diagram of the front desk. The model (left) imports nothing from the GUI. The controller implements `OrderBookListener`, calls the model and updates three view contracts; the real Swing or Tkinter views and the text views of the self-test both implement those contracts.

On the left is the **model**: `OrderBook` holds the orders and the rules and notifies `OrderBookListener`s; `LaundryOrder` is one order, which knows how to compute its chargeable weight and price; `ServiceType` and `OrderStatus` are enumerations (Chapter 9 introduced Java's `enum`; Python's `Enum` is its counterpart). In the middle is the **controller**, `DeskController`, which listens to the model and to the views. On the right are three **view contracts** — Java interfaces,

Python protocols — that say what the controller needs from a form, from an order list and from a summary line; below them, the real views and the text views of the self-test implement those contracts. Nothing in the model points to anything outside the model.

Each part of the window gets its own small interface, rather than one big `DeskView`, because the parts change independently: the waiting-room television of our story needs a list and nothing else. No class should be forced to depend on methods it does not use.

11.9.3 The model

Listing 11.11 is the model in Java. It has no `main`; the self-test and the checks use it.

Listing 11.11 `OrderBook.java`: the model of the front desk, with the order book, one order, the two enumerations and the listener interface. It imports nothing from Swing.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Locale;

// Model of the Profitina Laundry front desk: orders and the rules about them.
// It imports nothing from Swing and could serve a web page or a phone app.
public class OrderBook {
    public static final double MAX_KG = 50.0;
    private final List<LaundryOrder> orders = new ArrayList<>();
    private final List<OrderBookListener> listeners = new ArrayList<>();
    private int nextNumber = 1;

    public void addListener(OrderBookListener listener) {
        listeners.add(listener);
    }

    public void removeListener(OrderBookListener listener) {
        listeners.remove(listener);
    }

    private void fireChanged() {
        for (OrderBookListener listener : List.copyOf(listeners)) {
            listener.orderBookChanged(this);
        }
    }

    public LaundryOrder addOrder(String customer, ServiceType service,
                                 double weightKg, boolean express) {
        String name = customer == null ? "" : customer.strip();
        if (name.isEmpty()) {
            throw new IllegalArgumentException("customer name is required");
        }
        if (service == null) {
            throw new IllegalArgumentException("choose a service");
        }
        if (!(weightKg > 0 && weightKg <= MAX_KG)) {  // also rejects NaN
            throw new IllegalArgumentException("weight must be above 0 and at most 50 kg");
        }
        LaundryOrder order = new LaundryOrder(nextNumber++, name, service, weightKg,
express);
        orders.add(order);
        fireChanged();
        return order;
    }
```

Listing 11.11 (Continued) *OrderBook.java: the model of the front desk, with the order book, one order, the two enumerations and the listener interface. It imports nothing from Swing.*

```
    public void advance(int number) {
        LaundryOrder order = find(number);
        if (order.getStatus() == OrderStatus.PICKED_UP) {
            throw new IllegalStateException("order #" + number + " was already picked up");
        }
        order.setStatus(order.getStatus().next());
        fireChanged();
    }

    public LaundryOrder find(int number) {
        for (LaundryOrder order : orders) {
            if (order.getNumber() == number) {
                return order;
            }
        }
        throw new IllegalArgumentException("no order #" + number);
    }

    public List<LaundryOrder> getOrders() {
        return Collections.unmodifiableList(orders);
    }

    public int count() {
        return orders.size();
    }

    public int countWith(OrderStatus status) {
        int n = 0;
        for (LaundryOrder order : orders) {
            if (order.getStatus() == status) {
                n++;
            }
        }
        return n;
    }

    public long totalRevenue() {
        long total = 0;
        for (LaundryOrder order : orders) {
            total += order.price();
        }
        return total;
    }
```

Listing 11.11 (Continued) *OrderBook.java: the model of the front desk, with the order book, one order, the two enumerations and the listener interface. It imports nothing from Swing.*

```
    public static String rupiah(long amount) {
        return String.format(Locale.US, "Rp%,d", amount);
    }
}

enum ServiceType {
    WASH_FOLD("Wash & Fold", 7000, 3.0),
    WASH_IRON("Wash, Dry & Iron", 10000, 3.0),
    IRON_ONLY("Iron Only", 5000, 2.0);

    private final String label;
    private final long ratePerKg;
    private final double minimumKg;

    ServiceType(String label, long ratePerKg, double minimumKg) {
        this.label = label;
        this.ratePerKg = ratePerKg;
        this.minimumKg = minimumKg;
    }

    long ratePerKg() {
        return ratePerKg;
    }

    double minimumKg() {
        return minimumKg;
    }

    static ServiceType fromLabel(String label) {
        for (ServiceType type : values()) {
            if (type.label.equals(label)) {
                return type;
            }
        }
        return null;
    }

    @Override
    public String toString() {
        return label;
    }
}

enum OrderStatus {
    RECEIVED("Received"), WASHING("Washing"), READY("Ready"), PICKED_UP("Picked up");
```

Listing 11.11 (Continued) *OrderBook.java: the model of the front desk, with the order book, one order, the two enumerations and the listener interface. It imports nothing from Swing.*

```
    private final String label;

    OrderStatus(String label) {
        this.label = label;
    }

    OrderStatus next() {
        return values()[ordinal() + 1];  // PICKED_UP has no next: callers check first
    }

    @Override
    public String toString() {
        return label;
    }
}

final class LaundryOrder {
    private final int number;
    private final String customer;
    private final ServiceType service;
    private final double weightKg;
    private final boolean express;
    private OrderStatus status = OrderStatus.RECEIVED;

    LaundryOrder(int number, String customer, ServiceType service,
                 double weightKg, boolean express) {
        this.number = number;
        this.customer = customer;
        this.service = service;
        this.weightKg = weightKg;
        this.express = express;
    }

    int getNumber() {
        return number;
    }

    String getCustomer() {
        return customer;
    }

    ServiceType getService() {
        return service;
    }
```

Listing 11.11 (Continued) *OrderBook.java: the model of the front desk, with the order book, one order, the two enumerations and the listener interface. It imports nothing from Swing.*

```
    double getWeightKg() {
        return weightKg;
    }

    boolean isExpress() {
        return express;
    }

    OrderStatus getStatus() {
        return status;
    }

    void setStatus(OrderStatus status) {  // package-private: only OrderBook calls it
        this.status = status;
    }

    // At least the service minimum, rounded up to the next half kilogram.
    double chargeableKg() {
        return Math.ceil(Math.max(weightKg, service.minimumKg()) * 2) / 2;
    }

    long price() {
        long base = Math.round(chargeableKg() * service.ratePerKg());
        return express ? base * 3 / 2 : base;  // express costs half as much again
    }
}

interface OrderBookListener {
    void orderBookChanged(OrderBook book);
}
```

`addOrder` shows the order of operations every model method should follow: **validate, change, notify, return**. It refuses an empty name (after stripping spaces), a missing service, and a weight outside (0,50]. The weight test is written `!(weightKg > 0 && weightKg <= MAX_KG)` rather than the more obvious `weightKg <= 0 || weightKg > MAX_KG`, and the difference matters: the special value `NaN` ("not a number", which `Double.parseDouble("NaN")` happily returns) makes *every* comparison false, so the obvious test would let `NaN` through, while ours refuses it. Only when all checks have passed does the method create the order, add it to the list, and fire *one* event. If any check fails, the exception leaves the method before anything has changed and before anyone has been notified — so a refused order costs no order number and redraws no view.

`advance` follows the same pattern: find the order (or fail with `IllegalArgumentException`), refuse to advance a picked-up order with `IllegalStateException`, change the status, notify. The two exception types say different things: `IllegalArgumentException` means "you asked about something that does not exist or is malformed", and `IllegalStateException` means "your request is fine, but the object is not in a state where it can be granted". The controller catches them separately.

`getOrders` returns `Collections.unmodifiableList(orders)`, a read-only *view* of the internal list (Chapter 9). Anyone may read the orders; nobody outside `OrderBook` can add or remove one except through `addOrder`, which enforces the rules. `LaundryOrder`'s status setter is package-private and documented as "only `OrderBook` calls it". `fireChanged` iterates over `List.copyOf(listeners)`, a snapshot, for the reason Section 11.5.2 gave.

In `LaundryOrder`, `chargeableKg` is Equation (11.4) and `price` is Equation (11.5) followed by Equation (11.6). `ServiceType` stores each service's label, rate and minimum in the enum

constants themselves, so adding a fourth service is one line; `fromLabel` finds a constant by the text the form shows. `OrderStatus.next()` returns the following constant by position (`values()[ordinal() + 1]`); for `PICKED_UP` that would be past the end of the array, which is why `advance` checks first.

Listing 11.12 is the model in Python.

Listing 11.12 `order_book.py`: the front-desk model in Python. Like its Java twin, it imports nothing from Tkinter.

```
"""order_book.py: model of the Profitina Laundry front desk.

It imports nothing from Tkinter and could serve a web page or a phone app.
"""
import math
from enum import Enum

MAX_KG = 50.0


def rupiah(amount):
    return f"Rp{amount:,}"


class ServiceType(Enum):
    WASH_FOLD = ("Wash & Fold", 7000, 3.0)
    WASH_IRON = ("Wash, Dry & Iron", 10000, 3.0)
    IRON_ONLY = ("Iron Only", 5000, 2.0)

    def __init__(self, label, rate_per_kg, minimum_kg):
        self.label = label
        self.rate_per_kg = rate_per_kg
        self.minimum_kg = minimum_kg

    @classmethod
    def from_label(cls, label):
        for service in cls:
            if service.label == label:
                return service
        return None

    def __str__(self):
        return self.label


class OrderStatus(Enum):
    RECEIVED = "Received"
    WASHING = "Washing"
    READY = "Ready"
    PICKED_UP = "Picked up"

    def next(self):
        members = list(OrderStatus)
        return members[members.index(self) + 1]  # PICKED_UP has no next
```

Listing 11.12 (Continued) *order_book.py: the front-desk model in Python. Like its Java twin, it imports nothing from Tkinter.*

```
    def __str__(self):
        return self.value


class LaundryOrder:
    def __init__(self, number, customer, service, weight_kg, express):
        self.number = number
        self.customer = customer
        self.service = service
        self.weight_kg = weight_kg
        self.express = express
        self.status = OrderStatus.RECEIVED  # changed only by OrderBook

    def chargeable_kg(self):
        # At least the service minimum, rounded up to the next half kilogram.
        return math.ceil(max(self.weight_kg, self.service.minimum_kg) * 2) / 2

    def price(self):
        base = round(self.chargeable_kg() * self.service.rate_per_kg)
        return base * 3 // 2 if self.express else base  # express: half as much again


class OrderBook:
    def __init__(self):
        self._orders = []
        self._listeners = []
        self._next_number = 1

    def add_listener(self, listener):
        self._listeners.append(listener)

    def remove_listener(self, listener):
        self._listeners.remove(listener)

    def _fire_changed(self):
        for listener in list(self._listeners):
            listener(self)
```

Listing 11.12 (Continued) *order_book.py: the front-desk model in Python. Like its Java twin, it imports nothing from Tkinter.*

```
    def add_order(self, customer, service, weight_kg, express):
        name = (customer or "").strip()
        if not name:
            raise ValueError("customer name is required")
        if service is None:
            raise ValueError("choose a service")
        if not (0 < weight_kg <= MAX_KG):  # also rejects nan
            raise ValueError("weight must be above 0 and at most 50 kg")
        order = LaundryOrder(self._next_number, name, service, weight_kg, express)
        self._next_number += 1
        self._orders.append(order)
        self._fire_changed()
        return order

    def advance(self, number):
        order = self.find(number)
        if order.status is OrderStatus.PICKED_UP:
            raise RuntimeError(f"order #{number} was already picked up")
        order.status = order.status.next()
        self._fire_changed()

    def find(self, number):
        for order in self._orders:
            if order.number == number:
                return order
        raise ValueError(f"no order #{number}")

    def orders(self):
        return tuple(self._orders)  # a read-only snapshot

    def count(self):
        return len(self._orders)

    def count_with(self, status):
        return sum(1 for order in self._orders if order.status is status)

    def total_revenue(self):
        return sum(order.price() for order in self._orders)
```

The Python enum `ServiceType` shows a feature worth knowing: when an `Enum` member's value is a tuple, and the class defines `__init__`, Python calls `__init__` with the tuple's elements, so each member gets the attributes `label`, `rate_per_kg` and `minimum_kg`. The weight test `0 < weight_kg <= MAX_KG` is Python's chained comparison; `nan` fails it for the same reason as in Java. `orders()` returns a `tuple`, which cannot be changed, as the read-only view. `advance` raises `RuntimeError` where Java raised `IllegalStateException`, since Python has no exact equivalent; any exception type would do, as long as the controller catches the same one. And `round(...)` in `price` is safe even though Python rounds exact halves to the nearest even number: rates are multiples of 1,000 and chargeable weights multiples of 0.5, so the product is always a whole number.

11.9.4 The controller and the self-test

Listing 11.13 holds the three view interfaces, the controller, the text views and the `main` method.

Listing 11.13 `LaundryDesk.java`: the view contracts, the mediating controller, the text views of the self-test, and `main`.

```
import java.util.List;
import java.util.Locale;
import javax.swing.SwingUtilities;

public class LaundryDesk {
    public static void main(String[] args) {
        OrderBook book = new OrderBook();
        if (args.length > 0 && args[0].equals("--selftest")) {
            selfTest(book);
            return;
        }
        SwingUtilities.invokeLater(() -> {
            DeskWindow window = new DeskWindow("Profitina Laundry - Front Desk");
            new DeskController(book, window.form(), window.list(), window.summary());
            window.open();
        });
    }

    // A whole shift at the desk, typed and clicked by a script instead of a person.
    static void selfTest(OrderBook book) {
        TextFormView form = new TextFormView();
        TextListView list = new TextListView();
        TextSummaryView summary = new TextSummaryView();
        new DeskController(book, form, list, summary);

        form.fill("Sari Wulandari", "Wash & Fold", "4.2", false);
        form.clickAdd();
        form.fill("Budi", "Wash, Dry & Iron", "2,6", true);
        form.clickAdd();
        form.fill("   ", "Iron Only", "1.3", false);
        form.clickAdd();
        form.fill("Citra", "Iron Only", "abc", false);
        form.clickAdd();
        form.fill("Citra", "Iron Only", "75", false);
        form.clickAdd();
        form.fill("Citra", "Iron Only", "1.3", false);
        form.clickAdd();
        list.clickAdvance();
        list.select(2);
        for (int i = 0; i < 4; i++) {
            list.clickAdvance();
        }
```

Listing 11.13 (Continued) *LaundryDesk.java: the view contracts, the mediating controller, the text views of the self-test, and main.*

```
        boolean ok = book.count() == 3
                && book.totalRevenue() == 86_500
                && book.find(2).getStatus() == OrderStatus.PICKED_UP
                && book.countWith(OrderStatus.RECEIVED) == 2;
        if (!ok) {
            throw new AssertionError("selftest failed");
        }
        System.out.println("selftest passed: " + book.count() + " orders, "
                + OrderBook.rupiah(book.totalRevenue()));
    }
}

// What the controller needs from the order form, whatever draws it.
interface OrderFormView {
    String customerText();

    String serviceChoice();

    String weightText();

    boolean expressChecked();

    void onAdd(Runnable action);

    void clearForm();

    void showMessage(String message);
}

// What the controller needs from the order list.
interface OrderListView {
    void showOrders(List<LaundryOrder> orders);

    int selectedOrderNumber();  // 0 when nothing is selected

    void onAdvance(Runnable action);
}

// What the controller needs from the summary line.
interface SummaryView {
    void showSummary(int orders, int received, int ready, long revenue);
}
```

Listing 11.13 (Continued) *LaundryDesk.java: the view contracts, the mediating controller, the text views of the self-test, and main.*

```
// Controller: a mediator. It turns gestures into model calls and model changes
// into view updates; the model and the views never talk to each other directly.
class DeskController {
    private final OrderBook book;
    private final OrderFormView form;
    private final OrderListView list;
    private final SummaryView summary;

    DeskController(OrderBook book, OrderFormView form, OrderListView list, SummaryView
summary) {
        this.book = book;
        this.form = form;
        this.list = list;
        this.summary = summary;
        form.onAdd(this::addOrder);
        list.onAdvance(this::advanceSelected);
        book.addListener(this::refresh);
        refresh(book);
    }

    private void refresh(OrderBook changed) {
        list.showOrders(changed.getOrders());
        summary.showSummary(changed.count(), changed.countWith(OrderStatus.RECEIVED),
                changed.countWith(OrderStatus.READY), changed.totalRevenue());
    }

    void addOrder() {
        // The controller turns text into a number; the model judges the number.
        String text = form.weightText().strip().replace(',', '.');
        double kg;
        try {
            kg = Double.parseDouble(text);
        } catch (NumberFormatException e) {
            form.showMessage("Weight must be a number, such as 4.2");
            return;
        }
        ServiceType service = ServiceType.fromLabel(form.serviceChoice());
        try {
            LaundryOrder order = book.addOrder(form.customerText(), service, kg,
                    form.expressChecked());
            form.clearForm();
            form.showMessage("Added order #" + order.getNumber() + " for "
                    + order.getCustomer() + ": " + OrderBook.rupiah(order.price()));
        } catch (IllegalArgumentException e) {
            form.showMessage("Not added: " + e.getMessage());
        }
    }
```

Listing 11.13 (Continued) *LaundryDesk.java: the view contracts, the mediating controller, the text views of the self-test, and main.*

```
    void advanceSelected() {
        int number = list.selectedOrderNumber();
        if (number == 0) {
            form.showMessage("Select an order first.");
            return;
        }
        try {
            book.advance(number);
            form.showMessage("Order #" + number + " is now " +
book.find(number).getStatus());
        } catch (IllegalStateException e) {
            form.showMessage("Not changed: " + e.getMessage());
        }
    }
}

// Stand-in views for the self-test: each prints what its window part would show.
class TextFormView implements OrderFormView {
    private String customer = "";
    private String service = "Wash & Fold";
    private String weight = "";
    private boolean express;
    private Runnable addAction = () -> { };

    void fill(String customer, String service, String weight, boolean express) {
        this.customer = customer;
        this.service = service;
        this.weight = weight;
        this.express = express;
        System.out.printf("> type \"%s\", %s, \"%s\" kg%s%n",
                customer, service, weight, express ? ", Express" : "");
    }

    void clickAdd() {
        System.out.println("> click Add order");
        addAction.run();
    }

    @Override
    public String customerText() {
        return customer;
    }
```

Listing 11.13 (Continued) *LaundryDesk.java: the view contracts, the mediating controller, the text views of the self-test, and main.*

```
    @Override
    public String serviceChoice() {
        return service;
    }

    @Override
    public String weightText() {
        return weight;
    }

    @Override
    public boolean expressChecked() {
        return express;
    }

    @Override
    public void onAdd(Runnable action) {
        addAction = action;
    }

    @Override
    public void clearForm() {
        customer = "";
        weight = "";
        express = false;
    }

    @Override
    public void showMessage(String message) {
        System.out.println("  [message] " + message);
    }
}

class TextListView implements OrderListView {
    private int selected;
    private Runnable advanceAction = () -> { };

    void select(int number) {
        selected = number;
        System.out.println("> select order #" + number);
    }
```

Listing 11.13 (Continued) *LaundryDesk.java: the view contracts, the mediating controller, the text views of the self-test, and main.*

```
    void clickAdvance() {
        System.out.println("> click Advance status");
        advanceAction.run();
    }

    @Override
    public void showOrders(List<LaundryOrder> orders) {
        for (LaundryOrder o : orders) {
            System.out.println(String.format(Locale.US, "  [list] #%d %-15s %-17s %4.1f kg
%-3s %9s %s",
                    o.getNumber(), o.getCustomer(), o.getService(), o.getWeightKg(),
                    o.isExpress() ? "EX" : "", OrderBook.rupiah(o.price()),
o.getStatus()));
        }
    }

    @Override
    public int selectedOrderNumber() {
        return selected;
    }

    @Override
    public void onAdvance(Runnable action) {
        advanceAction = action;
    }
}

class TextSummaryView implements SummaryView {
    @Override
    public void showSummary(int orders, int received, int ready, long revenue) {
        String count = orders == 1 ? "1 order" : orders + " orders";
        System.out.println("  [summary] " + count + ", " + received + " received, "
                + ready + " ready, " + OrderBook.rupiah(revenue));
    }
}
```

The **view interfaces** are small and are written from the controller's point of view. `OrderFormView` offers four *queries* — what is in the customer field, which service is chosen, what is in the weight field, is *Express* ticked — one *registration*, `onAdd`, and two *commands*, `clearForm` and `showMessage`. `OrderListView` shows orders, reports which order is selected (0 for none) and registers the *Advance* action. `SummaryView` shows four numbers. Not one of these methods mentions Swing or Tkinter.

The **controller**'s constructor does all the wiring: it registers its two actions with the views, subscribes `refresh` to the model, and calls `refresh` once to draw the first picture. `refresh` is the pull style at work: whatever changed, it gives the list view the whole current list and the summary view all four numbers, recomputed from the model. The views therefore never need to know *what* changed.

`addOrder` shows the division of labor between controller and model in its purest form. The controller's job is to turn *text* into *values*: it strips the weight text, replaces a comma by a point, and parses a `double`; if parsing fails, it shows a message and stops, because a string such as `abc` is not a weight at all and the model should never see it. It turns the service label into a `ServiceType`. Then it hands values to the model and lets the model judge them: whether 75 kg is acceptable is a *business* rule, and it lives in the model, where the unit tests can reach it. If the model refuses, the controller shows the model's own explanation prefixed with `Not added:`. If it accepts, the controller clears the form and reports the number and the price. The controller never computes a

price and never checks a weight range. `advanceSelected` follows the same pattern for the second button.

`main` either runs the self-test or builds the real window on the event dispatch thread. The **self-test** is a whole shift at the desk: it types six orders, three of which are wrong in different ways, clicks *Advance status* once with nothing selected, then selects order 2 and advances it four times — one more than the state machine allows. The text views print what the window would show. `java LaundryDesk --selftest` prints:

```
  [summary] 0 orders, 0 received, 0 ready, Rp0
> type "Sari Wulandari", Wash & Fold, "4.2" kg
> click Add order
  [list] #1 Sari Wulandari  Wash & Fold        4.2 kg      Rp31,500 Received
  [summary] 1 order, 1 received, 0 ready, Rp31,500
  [message] Added order #1 for Sari Wulandari: Rp31,500
> type "Budi", Wash, Dry & Iron, "2,6" kg, Express
> click Add order
  [list] #1 Sari Wulandari  Wash & Fold        4.2 kg      Rp31,500 Received
  [list] #2 Budi            Wash, Dry & Iron   2.6 kg EX   Rp45,000 Received
  [summary] 2 orders, 2 received, 0 ready, Rp76,500
  [message] Added order #2 for Budi: Rp45,000
> type "   ", Iron Only, "1.3" kg
> click Add order
  [message] Not added: customer name is required
> type "Citra", Iron Only, "abc" kg
> click Add order
  [message] Weight must be a number, such as 4.2
> type "Citra", Iron Only, "75" kg
> click Add order
  [message] Not added: weight must be above 0 and at most 50 kg
> type "Citra", Iron Only, "1.3" kg
> click Add order
  [list] #1 Sari Wulandari  Wash & Fold        4.2 kg      Rp31,500 Received
  [list] #2 Budi            Wash, Dry & Iron   2.6 kg EX   Rp45,000 Received
  [list] #3 Citra           Iron Only          1.3 kg      Rp10,000 Received
  [summary] 3 orders, 3 received, 0 ready, Rp86,500
  [message] Added order #3 for Citra: Rp10,000
> click Advance status
  [message] Select an order first.
> select order #2
> click Advance status
  [list] #1 Sari Wulandari  Wash & Fold        4.2 kg      Rp31,500 Received
  [list] #2 Budi            Wash, Dry & Iron   2.6 kg EX   Rp45,000 Washing
  [list] #3 Citra           Iron Only          1.3 kg      Rp10,000 Received
  [summary] 3 orders, 2 received, 0 ready, Rp86,500
  [message] Order #2 is now Washing
> click Advance status
  [list] #1 Sari Wulandari  Wash & Fold        4.2 kg      Rp31,500 Received
  [list] #2 Budi            Wash, Dry & Iron   2.6 kg EX   Rp45,000 Ready
  [list] #3 Citra           Iron Only          1.3 kg      Rp10,000 Received
  [summary] 3 orders, 2 received, 1 ready, Rp86,500
  [message] Order #2 is now Ready
> click Advance status
  [list] #1 Sari Wulandari  Wash & Fold        4.2 kg      Rp31,500 Received
  [list] #2 Budi            Wash, Dry & Iron   2.6 kg EX   Rp45,000 Picked up
  [list] #3 Citra           Iron Only          1.3 kg      Rp10,000 Received
  [summary] 3 orders, 2 received, 0 ready, Rp86,500
  [message] Order #2 is now Picked up
> click Advance status
  [message] Not changed: order #2 was already picked up
selftest passed: 3 orders, Rp86,500
```

Each valid `> click Add order` is followed by the list and summary lines — printed by `refresh` during the model's notification, *inside* `book.addOrder` — and only then by the form's message. The

three refused orders produce only a message: the model did not change, so no view redrew, and no order number was used, so Citra's valid order is #3. Budi's weight was typed `2,6` and understood as 2.6. The four clicks of *Advance status* move order #2 through its states, and the summary's counts change with them — 2 received and 1 ready while Budi's clothes wait on the shelf, 0 ready once he has collected them — until the fourth click is refused with the model's message. At the end the self-test checks the count, the revenue of Rp86,500 that we computed by hand, the status of order #2 and the number of received orders.

Figure 11.13 draws the first successful click on **Add order** as a sequence diagram, and it is worth tracing against the output.

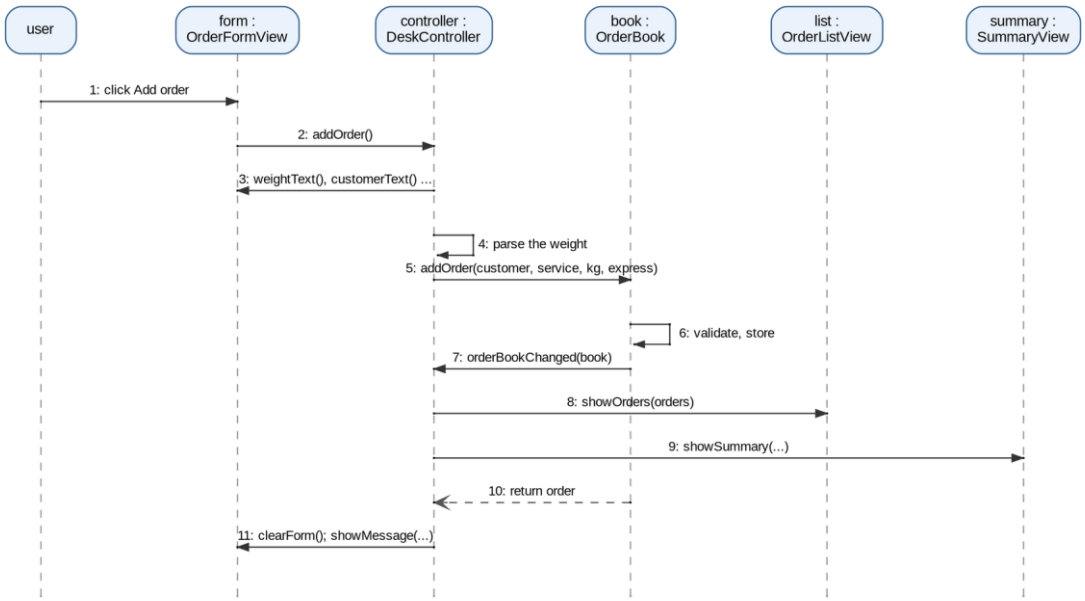


Figure 11.13 Sequence diagram of a click on *Add order*. The form reports the click to the controller, which reads the form, parses the weight and calls the model; the model validates, stores and notifies the controller, which refreshes the list and the summary; after the model returns, the controller clears the form and shows a message.

Messages 1–2 are the gesture, 3–4 the controller reading and parsing the form, 5–6 the command and the model's validation. Messages 7–9, the notification and the redraw, happen *inside* message 5, before the model returns (10); message 11 updates the form.

11.9.5 The Swing views

Listing 11.14 contains the real views: three Swing components, each implementing one view interface, and the `DeskWindow` class that assembles them.

Listing 11.14 `DeskWindow.java`: the Swing views of the front desk. `OrderFormPanel`, `OrderTablePanel` and `SummaryBar` implement the three view contracts; `DeskWindow` places them in a frame.

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.util.List;
import java.util.Locale;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.ListSelectionModel;
import javax.swing.table.DefaultTableModel;

// The Swing views of the front desk, assembled in one window.
public class DeskWindow {
    private final JFrame frame;
    private final OrderFormPanel form = new OrderFormPanel();
    private final OrderTablePanel list = new OrderTablePanel();
    private final SummaryBar summary = new SummaryBar();

    public DeskWindow(String title) {
        frame = new JFrame(title);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(form, BorderLayout.NORTH);
        frame.add(list, BorderLayout.CENTER);
        frame.add(summary, BorderLayout.SOUTH);
        frame.setSize(760, 420);
    }

    OrderFormView form() {
        return form;
    }

    OrderListView list() {
        return list;
    }

    SummaryView summary() {
        return summary;
    }
```

Listing 11.14 (Continued) *DeskWindow.java: the Swing views of the front desk. OrderFormPanel, OrderTablePanel and SummaryBar implement the three view contracts; DeskWindow places them in a frame.*

```
    void open() {
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}

class OrderFormPanel extends JPanel implements OrderFormView {
    private final JTextField customer = new JTextField(14);
    private final JComboBox<String> service = new JComboBox<>();
    private final JTextField weight = new JTextField(5);
    private final JCheckBox express = new JCheckBox("Express");
    private final JButton add = new JButton("Add order");
    private final JLabel message = new JLabel("Ready.");

    OrderFormPanel() {
        super(new BorderLayout());
        for (ServiceType type : ServiceType.values()) {
            service.addItem(type.toString());
        }
        // Two rows: one FlowLayout row at this width would hide "Add order".
        JPanel rows = new JPanel(new GridLayout(2, 1));
        JPanel first = new JPanel(new FlowLayout(FlowLayout.LEFT));
        first.add(new JLabel("Customer"));
        first.add(customer);
        first.add(new JLabel("Service"));
        first.add(service);
        JPanel second = new JPanel(new FlowLayout(FlowLayout.LEFT));
        second.add(new JLabel("Weight (kg)"));
        second.add(weight);
        second.add(express);
        second.add(add);
        rows.add(first);
        rows.add(second);
        add(rows, BorderLayout.CENTER);
        message.setBorder(BorderFactory.createEmptyBorder(0, 10, 5, 10));
        add(message, BorderLayout.SOUTH);
    }

    @Override
    public String customerText() {
        return customer.getText();
    }
```

Listing 11.14 (Continued) *DeskWindow.java: the Swing views of the front desk. OrderFormPanel, OrderTablePanel and SummaryBar implement the three view contracts; DeskWindow places them in a frame.*

```
    @Override
    public String serviceChoice() {
        return (String) service.getSelectedItem();
    }

    @Override
    public String weightText() {
        return weight.getText();
    }

    @Override
    public boolean expressChecked() {
        return express.isSelected();
    }

    @Override
    public void onAdd(Runnable action) {
        add.addActionListener(e -> action.run());
    }

    @Override
    public void clearForm() {
        customer.setText("");
        weight.setText("");
        express.setSelected(false);
        customer.requestFocusInWindow();
    }

    @Override
    public void showMessage(String text) {
        message.setText(text);
    }
}

class OrderTablePanel extends JPanel implements OrderListView {
    private static final String[] COLUMNS =
            {"No.", "Customer", "Service", "kg", "Express", "Price", "Status"};
    private final DefaultTableModel rows = new DefaultTableModel(COLUMNS, 0) {
        @Override
        public boolean isCellEditable(int row, int column) {
            return false;  // the table only displays; changes go through the controller
        }
    };
    private final JTable table = new JTable(rows);
    private final JButton advance = new JButton("Advance status");
```

Listing 11.14 (Continued) *DeskWindow.java: the Swing views of the front desk. OrderFormPanel, OrderTablePanel and SummaryBar implement the three view contracts; DeskWindow places them in a frame.*

```
    OrderTablePanel() {
        super(new BorderLayout());
        table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        add(new JScrollPane(table), BorderLayout.CENTER);
        JPanel buttons = new JPanel(new FlowLayout(FlowLayout.RIGHT));
        buttons.add(advance);
        add(buttons, BorderLayout.SOUTH);
    }

    @Override
    public void showOrders(List<LaundryOrder> orders) {
        int keep = selectedOrderNumber();
        rows.setRowCount(0);
        for (LaundryOrder o : orders) {
            rows.addRow(new Object[] {o.getNumber(), o.getCustomer(), o.getService(),
                String.format(Locale.US, "%.1f", o.getWeightKg()),
                o.isExpress() ? "yes" : "", OrderBook.rupiah(o.price()), o.getStatus()});
            if (o.getNumber() == keep) {
                table.setRowSelectionInterval(rows.getRowCount() - 1, rows.getRowCount() -
1);
            }
        }
    }

    @Override
    public int selectedOrderNumber() {
        int row = table.getSelectedRow();
        return row < 0 ? 0 : (Integer) rows.getValueAt(row, 0);
    }

    @Override
    public void onAdvance(Runnable action) {
        advance.addActionListener(e -> action.run());
    }
}

class SummaryBar extends JLabel implements SummaryView {
    SummaryBar() {
        setBorder(BorderFactory.createEmptyBorder(5, 10, 5, 10));
    }
```

Listing 11.14 (Continued) *DeskWindow.java: the Swing views of the front desk. OrderFormPanel, OrderTablePanel and SummaryBar implement the three view contracts; DeskWindow places them in a frame.*

```
    @Override
    public void showSummary(int orders, int received, int ready, long revenue) {
        String count = orders == 1 ? "1 order" : orders + " orders";
        setText(count + "  |  " + received + " received  |  " + ready
                + " ready for pickup  |  total " + OrderBook.rupiah(revenue));
    }
}
```

Every method of these classes is either building widgets or moving data between a widget and a parameter; there is no `if` about business in the whole file. `OrderFormPanel` builds two rows with `FlowLayout`, in a `GridLayout` of two rows: the customer field and the service box in the first, the weight field, the *Express* check box and the **Add order** button in the second, and the message label below. The comment above the rows records a lesson learned while building this program: the first version put all eight components in *one* `FlowLayout` row. `FlowLayout` wraps components that do not fit onto a new line, but the `NORTH` region of a `BorderLayout` gives the panel only the height of *one* line — so at 760 pixels wide, the **Add order** button wrapped onto a second line that

was never drawn. The program compiled, the self-test passed, and the button was invisible. Only looking at the window revealed it. It is the clearest possible demonstration of what the self-test can and cannot see.

`OrderTablePanel` shows the orders in a `JTable` whose `DefaultTableModel` refuses edits: changes go through the controller. `showOrders` restores the previous selection after rebuilding the rows, so the clerk can click *Advance status* repeatedly without reselecting. `SummaryBar` is a `JLabel` with a margin.

Run `java LaundryDesk` on a computer with a screen and the window looks like this after the three valid orders of the self-test have been added, order 2 has been advanced twice, and the clerk has typed an invalid weight for a fourth customer:

```
+-------------------------------------------------------------------------------+
| Profitina Laundry - Front Desk                                          _ o x |
+-------------------------------------------------------------------------------+
| Customer [Dewi          ]  Service [Iron Only       v]                        |
| Weight (kg) [abc  ]  [ ] Express  [ Add order ]                               |
|  Weight must be a number, such as 4.2                                         |
| +-----+----------------+------------------+-----+---------+----------+-----------+ |
| | No. | Customer       | Service          | kg  | Express | Price    | Status    | |
| +-----+----------------+------------------+-----+---------+----------+-----------+ |
| | 1   | Sari Wulandari | Wash & Fold      | 4.2 |         | Rp31,500 | Received  | |
| |#2###|#Budi###########|#Wash, Dry & Iron#|#2.6#|#yes#####|#Rp45,000#|#Ready#####| |
| | 3   | Citra          | Iron Only        | 1.3 |         | Rp10,000 | Received  | |
| |     |                |                  |     |         |          |           | |
| +-----+----------------+------------------+-----+---------+----------+-----------+ |
|                                                          [ Advance status ]   |
| 3 orders  |  2 received  |  1 ready for pickup  |  total Rp86,500              |
+-------------------------------------------------------------------------------+
```

The window is 760 by 420 pixels and opens in the center of the screen. At the top is the form, in two rows, with the message line beneath it; it reads *Ready.* when the program starts. The table fills the middle and grows when the window is enlarged; the selected row (drawn with `#` in the sketch) is highlighted in the system's selection color. The **Advance status** button sits at the bottom right of the table, and the summary line runs along the bottom of the window.

11.9.6 The same desk in Python

In Python, the controller and the text views live in `laundry_desk.py` (**Listing 11.15**) and the Tkinter views in a separate module, `tk_desk_views.py` (**Listing 11.16**). The separation into two files is deliberate: `laundry_desk.py` imports the Tkinter module only in the branch of `main` that opens a window, so the self-test never loads Tkinter at all.

Listing 11.15 `laundry_desk.py`: the view protocols, the controller, the text views of the self-test, and `main`.

```
"""laundry_desk.py: the Profitina Laundry front desk as model, views and controller."""
import sys
from typing import Callable, Protocol, Sequence

from order_book import LaundryOrder, OrderBook, OrderStatus, ServiceType, rupiah


class OrderFormView(Protocol):
    """What the controller needs from the order form, whatever draws it."""
    def customer_text(self) -> str: ...
    def service_choice(self) -> str: ...
    def weight_text(self) -> str: ...
    def express_checked(self) -> bool: ...
    def on_add(self, action: Callable[[], None]) -> None: ...
    def clear_form(self) -> None: ...
    def show_message(self, message: str) -> None: ...


class OrderListView(Protocol):
    """What the controller needs from the order list."""
    def show_orders(self, orders: Sequence[LaundryOrder]) -> None: ...
    def selected_order_number(self) -> int: ...  # 0 when nothing is selected
    def on_advance(self, action: Callable[[], None]) -> None: ...


class SummaryView(Protocol):
    """What the controller needs from the summary line."""
    def show_summary(self, orders: int, received: int, ready: int,
                     revenue: int) -> None: ...


class DeskController:
    """Controller: a mediator. It turns gestures into model calls and model
    changes into view updates; the model and the views never talk directly."""

    def __init__(self, book: OrderBook, form: OrderFormView,
                 order_list: OrderListView, summary: SummaryView):
        self._book = book
        self._form = form
        self._list = order_list
        self._summary = summary
        form.on_add(self.add_order)
        order_list.on_advance(self.advance_selected)
        book.add_listener(self._refresh)
        self._refresh(book)
```

Listing 11.15 (Continued) *laundry_desk.py: the view protocols, the controller, the text views of the self-test, and main.*

```
    def _refresh(self, changed):
        self._list.show_orders(changed.orders())
        self._summary.show_summary(changed.count(),
                                   changed.count_with(OrderStatus.RECEIVED),
                                   changed.count_with(OrderStatus.READY),
                                   changed.total_revenue())

    def add_order(self):
        # The controller turns text into a number; the model judges the number.
        text = self._form.weight_text().strip().replace(",", ".")
        try:
            kg = float(text)
        except ValueError:
            self._form.show_message("Weight must be a number, such as 4.2")
            return
        service = ServiceType.from_label(self._form.service_choice())
        try:
            order = self._book.add_order(self._form.customer_text(), service, kg,
                                         self._form.express_checked())
        except ValueError as e:
            self._form.show_message(f"Not added: {e}")
            return
        self._form.clear_form()
        self._form.show_message(f"Added order #{order.number} for "
                                f"{order.customer}: {rupiah(order.price())}")

    def advance_selected(self):
        number = self._list.selected_order_number()
        if number == 0:
            self._form.show_message("Select an order first.")
            return
        try:
            self._book.advance(number)
        except RuntimeError as e:
            self._form.show_message(f"Not changed: {e}")
            return
        self._form.show_message(f"Order #{number} is now {self._book.find(number).status}")


class TextFormView:
    """Stand-in form for the self-test: it prints what the window would show."""
```

Listing 11.15 (Continued) *laundry_desk.py: the view protocols, the controller, the text views of the self-test, and main.*

```
    def __init__(self):
        self._customer, self._service, self._weight = "", "Wash & Fold", ""
        self._express = False
        self._add = lambda: None

    def fill(self, customer, service, weight, express):
        self._customer, self._service = customer, service
        self._weight, self._express = weight, express
        print(f'> type "{customer}", {service}, "{weight}" kg'
              + (", Express" if express else ""))

    def click_add(self):
        print("> click Add order")
        self._add()

    def customer_text(self):
        return self._customer

    def service_choice(self):
        return self._service

    def weight_text(self):
        return self._weight

    def express_checked(self):
        return self._express

    def on_add(self, action):
        self._add = action

    def clear_form(self):
        self._customer, self._weight, self._express = "", "", False

    def show_message(self, message):
        print("  [message] " + message)


class TextListView:
    def __init__(self):
        self._selected = 0
        self._advance = lambda: None

    def select(self, number):
        self._selected = number
        print(f"> select order #{number}")
```

Listing 11.15 (Continued) *laundry_desk.py: the view protocols, the controller, the text views of the self-test, and main.*

```
    def click_advance(self):
        print("> click Advance status")
        self._advance()

    def show_orders(self, orders):
        for o in orders:
            print(f"  [list] #{o.number} {o.customer:<15} {str(o.service):<17} "
                  f"{o.weight_kg:4.1f} kg {'EX' if o.express else '':<3} "
                  f"{rupiah(o.price()):>9} {o.status}")

    def selected_order_number(self):
        return self._selected

    def on_advance(self, action):
        self._advance = action


class TextSummaryView:
    def show_summary(self, orders, received, ready, revenue):
        count = "1 order" if orders == 1 else f"{orders} orders"
        print(f"  [summary] {count}, {received} received, {ready} ready, "
              f"{rupiah(revenue)}")


def self_test(book):
    """A whole shift at the desk, typed and clicked by a script instead of a person."""
    form, order_list, summary = TextFormView(), TextListView(), TextSummaryView()
    DeskController(book, form, order_list, summary)

    form.fill("Sari Wulandari", "Wash & Fold", "4.2", False)
    form.click_add()
    form.fill("Budi", "Wash, Dry & Iron", "2,6", True)
    form.click_add()
    form.fill("   ", "Iron Only", "1.3", False)
    form.click_add()
    form.fill("Citra", "Iron Only", "abc", False)
    form.click_add()
    form.fill("Citra", "Iron Only", "75", False)
    form.click_add()
    form.fill("Citra", "Iron Only", "1.3", False)
    form.click_add()
    order_list.click_advance()
    order_list.select(2)
    for _ in range(4):
        order_list.click_advance()
```

Listing 11.15 (Continued) *laundry_desk.py: the view protocols, the controller, the text views of the self-test, and main.*

```
    assert (book.count() == 3 and book.total_revenue() == 86_500
            and book.find(2).status is OrderStatus.PICKED_UP
            and book.count_with(OrderStatus.RECEIVED) == 2), "selftest failed"
    print(f"selftest passed: {book.count()} orders, {rupiah(book.total_revenue())}")


def main(args):
    book = OrderBook()
    if args[:1] == ["--selftest"]:
        self_test(book)
        return
    from tk_desk_views import TkDeskWindow  # the GUI is loaded only when needed
    window = TkDeskWindow("Profitina Laundry - Front Desk")
    DeskController(book, window.form, window.list, window.summary)
    window.open()


if __name__ == "__main__":
    main(sys.argv[1:])
```

`python3 laundry_desk.py --selftest` prints the same transcript as Java, line for line:

```
  [summary] 0 orders, 0 received, 0 ready, Rp0
> type "Sari Wulandari", Wash & Fold, "4.2" kg
> click Add order
  [list] #1 Sari Wulandari  Wash & Fold        4.2 kg      Rp31,500 Received
  [summary] 1 order, 1 received, 0 ready, Rp31,500
  [message] Added order #1 for Sari Wulandari: Rp31,500
> type "Budi", Wash, Dry & Iron, "2,6" kg, Express
> click Add order
  [list] #1 Sari Wulandari  Wash & Fold        4.2 kg      Rp31,500 Received
  [list] #2 Budi            Wash, Dry & Iron   2.6 kg EX   Rp45,000 Received
  [summary] 2 orders, 2 received, 0 ready, Rp76,500
  [message] Added order #2 for Budi: Rp45,000
> type "   ", Iron Only, "1.3" kg
> click Add order
  [message] Not added: customer name is required
> type "Citra", Iron Only, "abc" kg
> click Add order
  [message] Weight must be a number, such as 4.2
> type "Citra", Iron Only, "75" kg
> click Add order
  [message] Not added: weight must be above 0 and at most 50 kg
> type "Citra", Iron Only, "1.3" kg
> click Add order
  [list] #1 Sari Wulandari  Wash & Fold        4.2 kg      Rp31,500 Received
  [list] #2 Budi            Wash, Dry & Iron   2.6 kg EX   Rp45,000 Received
  [list] #3 Citra           Iron Only          1.3 kg      Rp10,000 Received
  [summary] 3 orders, 3 received, 0 ready, Rp86,500
  [message] Added order #3 for Citra: Rp10,000
> click Advance status
  [message] Select an order first.
> select order #2
> click Advance status
  [list] #1 Sari Wulandari  Wash & Fold        4.2 kg      Rp31,500 Received
  [list] #2 Budi            Wash, Dry & Iron   2.6 kg EX   Rp45,000 Washing
  [list] #3 Citra           Iron Only          1.3 kg      Rp10,000 Received
  [summary] 3 orders, 2 received, 0 ready, Rp86,500
  [message] Order #2 is now Washing
> click Advance status
  [list] #1 Sari Wulandari  Wash & Fold        4.2 kg      Rp31,500 Received
  [list] #2 Budi            Wash, Dry & Iron   2.6 kg EX   Rp45,000 Ready
  [list] #3 Citra           Iron Only          1.3 kg      Rp10,000 Received
  [summary] 3 orders, 2 received, 1 ready, Rp86,500
  [message] Order #2 is now Ready
> click Advance status
  [list] #1 Sari Wulandari  Wash & Fold        4.2 kg      Rp31,500 Received
  [list] #2 Budi            Wash, Dry & Iron   2.6 kg EX   Rp45,000 Picked up
  [list] #3 Citra           Iron Only          1.3 kg      Rp10,000 Received
  [summary] 3 orders, 2 received, 0 ready, Rp86,500
  [message] Order #2 is now Picked up
> click Advance status
  [message] Not changed: order #2 was already picked up
selftest passed: 3 orders, Rp86,500
```

The view contracts are written as **protocols**, `class OrderFormView(Protocol)`, from the module `typing` [10]. A protocol lists method signatures; any class that has those methods *is* an `OrderFormView` without declaring it — it writes down Python's duck typing. Python does not check protocols at run time; a static checker such as *mypy* does, and a reader sees at a glance what the controller expects. Otherwise the controller is a translation of Java's: `float(text)` raises `ValueError` for text that is not a number, `except ValueError as e` catches both that and the model's refusals, and `except RuntimeError` catches the refusal to advance a picked-up order.

Listing 11.16 `tk_desk_views.py`: the Tkinter views of the front desk.

```
"""tk_desk_views.py: the Tkinter views of the front desk, assembled in one window."""
import tkinter as tk
from tkinter import ttk

from order_book import ServiceType, rupiah


class OrderFormFrame(ttk.Frame):
    def __init__(self, parent):
        super().__init__(parent, padding=5)
        self._customer = tk.StringVar()
        self._service = tk.StringVar(value=ServiceType.WASH_FOLD.label)
        self._weight = tk.StringVar()
        self._express = tk.BooleanVar()
        self._message = tk.StringVar(value="Ready.")
        ttk.Label(self, text="Customer").grid(row=0, column=0, sticky="w")
        self._customer_entry = ttk.Entry(self, textvariable=self._customer, width=18)
        self._customer_entry.grid(row=0, column=1, sticky="w")
        ttk.Label(self, text="Service").grid(row=0, column=2, sticky="w", padx=(10, 0))
        ttk.Combobox(self, textvariable=self._service, state="readonly", width=16,
                     values=[s.label for s in ServiceType]).grid(row=0, column=3)
        ttk.Label(self, text="Weight (kg)").grid(row=1, column=0, sticky="w")
        ttk.Entry(self, textvariable=self._weight, width=6).grid(row=1, column=1,
sticky="w")
        ttk.Checkbutton(self, text="Express", variable=self._express).grid(
            row=1, column=2, sticky="w", padx=(10, 0))
        self._add = ttk.Button(self, text="Add order")
        self._add.grid(row=1, column=3, sticky="e")
        ttk.Label(self, textvariable=self._message).grid(
            row=2, column=0, columnspan=4, sticky="w", pady=(5, 0))

    def customer_text(self):
        return self._customer.get()

    def service_choice(self):
        return self._service.get()

    def weight_text(self):
        return self._weight.get()

    def express_checked(self):
        return self._express.get()

    def on_add(self, action):
        self._add.config(command=action)
```

Listing 11.16 (Continued) *tk_desk_views.py: the Tkinter views of the front desk.*

```
    def clear_form(self):
        self._customer.set("")
        self._weight.set("")
        self._express.set(False)
        self._customer_entry.focus_set()

    def show_message(self, message):
        self._message.set(message)


class OrderTableFrame(ttk.Frame):
    WIDTHS = {"No.": 50, "Customer": 140, "Service": 140, "kg": 60,
              "Express": 70, "Price": 100, "Status": 100}

    def __init__(self, parent):
        super().__init__(parent, padding=5)
        self._table = ttk.Treeview(self, columns=tuple(self.WIDTHS), show="headings",
                                   selectmode="browse", height=10)
        for name, width in self.WIDTHS.items():
            self._table.heading(name, text=name)
            self._table.column(name, width=width)
        self._table.pack(fill="both", expand=True)
        self._advance = ttk.Button(self, text="Advance status")
        self._advance.pack(anchor="e", pady=(5, 0))

    def show_orders(self, orders):
        keep = self.selected_order_number()
        self._table.delete(*self._table.get_children())
        for o in orders:
            # The order number is the row id, so the selection survives a refresh.
            self._table.insert("", "end", iid=str(o.number), values=(
                o.number, o.customer, str(o.service), f"{o.weight_kg:.1f}",
                "yes" if o.express else "", rupiah(o.price()), str(o.status)))
        if keep:
            self._table.selection_set(str(keep))

    def selected_order_number(self):
        selected = self._table.selection()
        return int(selected[0]) if selected else 0

    def on_advance(self, action):
        self._advance.config(command=action)
```

Listing 11.16 (Continued) *tk_desk_views.py: the Tkinter views of the front desk.*

```
class SummaryBar(ttk.Label):
    def __init__(self, parent):
        super().__init__(parent, padding=(10, 5))

    def show_summary(self, orders, received, ready, revenue):
        count = "1 order" if orders == 1 else f"{orders} orders"
        self.config(text=f"{count}  |  {received} received  |  {ready} ready "
                         f"for pickup  |  total {rupiah(revenue)}")


class TkDeskWindow:
    def __init__(self, title):
        self._root = tk.Tk()
        self._root.title(title)
        self.form = OrderFormFrame(self._root)
        self.form.pack(fill="x")
        self.list = OrderTableFrame(self._root)
        self.list.pack(fill="both", expand=True)
        self.summary = SummaryBar(self._root)
        self.summary.pack(fill="x")

    def open(self):
        self._root.mainloop()
```

The Tkinter views use the *themed* widgets of `tkinter.ttk`, which look native on each operating system. The form is a grid of three rows. Each input is connected to a Tkinter variable — `StringVar` for text, `BooleanVar` for the check box — so the queries of the view contract simply call `get()`. The service is a read-only `Combobox` filled with the service labels. The order list is a `ttk.Treeview` with `show="headings"`, which turns the tree widget into a table; each row's *item id* is the order number as a string, so `showOrders` can restore the selection with `selection_set(str(keep))` and `selected_order_number` can read the number back with `int(...)`. The window has the same layout as the Swing one: the form at the top with the message line, the table in the middle with **Advance status** below it on the right, and the summary line at the bottom.

11.10 Unit Tests for the Front Desk Model

The self-test of the front desk is a scenario; the model deserves proper unit tests too, which check each rule separately. **Listing 11.17** checks twenty facts in plain Java.

Listing 11.17 `OrderBookCheck.java`: twenty checks of the front-desk model, including prices, refusals, the state machine, the read-only list and unsubscribing.

```
import java.util.Objects;

public class OrderBookCheck {
    private static int passed;
    private static int failed;

    public static void main(String[] args) {
        OrderBook book = new OrderBook();
        int[] events = {0};
        OrderBookListener counter = b -> events[0]++;
        book.addListener(counter);

        LaundryOrder sari = book.addOrder("  Sari Wulandari ", ServiceType.WASH_FOLD, 4.2,
false);
        LaundryOrder budi = book.addOrder("Budi", ServiceType.WASH_IRON, 2.6, true);
        LaundryOrder citra = book.addOrder("Citra", ServiceType.IRON_ONLY, 1.3, false);
        check("name is trimmed", "Sari Wulandari", sari.getCustomer());
        check("4.2 kg is charged as 4.5 kg", 4.5, sari.chargeableKg());
        check("wash and fold 4.2 kg", 31_500L, sari.price());
        check("minimum 3 kg, then express x 3/2", 45_000L, budi.price());
        check("iron only has a 2 kg minimum", 10_000L, citra.price());
        check("numbers count up", 3, citra.getNumber());
        check("total revenue", 86_500L, book.totalRevenue());
        check("one event per order", 3, events[0]);

        expectRefused("blank name", () -> book.addOrder("   ", ServiceType.WASH_FOLD, 3,
false));
        expectRefused("zero weight", () -> book.addOrder("Dewi", ServiceType.WASH_FOLD, 0,
false));
        expectRefused("too heavy", () -> book.addOrder("Dewi", ServiceType.WASH_FOLD, 50.5,
false));
        expectRefused("not a number", () -> book.addOrder("Dewi", ServiceType.WASH_FOLD,
Double.NaN, false));
        expectRefused("no service", () -> book.addOrder("Dewi", null, 3, false));
        check("refused orders change nothing", 3, book.count());
        check("refused orders fire no event", 3, events[0]);

        for (int i = 0; i < 3; i++) {
            book.advance(2);
        }
        check("three steps reach Picked up", OrderStatus.PICKED_UP, budi.getStatus());
        try {
            book.advance(2);
            check("a fourth step is refused", "exception", "no exception");
        } catch (IllegalStateException e) {
            check("a fourth step is refused", "order #2 was already picked up",
e.getMessage());
        }
        check("counting by status", 2, book.countWith(OrderStatus.RECEIVED));
```

Listing 11.17 (Continued) *OrderBookCheck.java: twenty checks of the front-desk model, including prices, refusals, the state machine, the read-only list and unsubscribing.*

```
        try {
            book.getOrders().clear();
            check("the list is read-only", "exception", "no exception");
        } catch (UnsupportedOperationException e) {
            check("the list is read-only", 3, book.count());
        }

        book.removeListener(counter);
        book.advance(1);
        check("a removed listener hears nothing", 6, events[0]);

        System.out.println(passed + " of " + (passed + failed) + " checks passed");
        if (failed > 0) {
            System.exit(1);
        }
    }

    static void expectRefused(String name, Runnable action) {
        try {
            action.run();
            check(name + " is refused", "exception", "no exception");
        } catch (IllegalArgumentException e) {
            check(name + " is refused", true, true);
        }
    }

    static void check(String name, Object expected, Object actual) {
        if (Objects.equals(expected, actual)) {
            passed++;
            System.out.println("PASS " + name);
        } else {
            failed++;
            System.out.println("FAIL " + name + ": expected " + expected + ", got " +
actual);
        }
    }
}
```

`java OrderBookCheck` prints:

```
PASS name is trimmed
PASS 4.2 kg is charged as 4.5 kg
PASS wash and fold 4.2 kg
PASS minimum 3 kg, then express x 3/2
PASS iron only has a 2 kg minimum
PASS numbers count up
PASS total revenue
PASS one event per order
PASS blank name is refused
PASS zero weight is refused
PASS too heavy is refused
PASS not a number is refused
PASS no service is refused
PASS refused orders change nothing
PASS refused orders fire no event
PASS three steps reach Picked up
PASS a fourth step is refused
PASS counting by status
PASS the list is read-only
PASS a removed listener hears nothing
20 of 20 checks passed
```

The checks follow the rules of Section 11.9.1 one by one. The prices are the three we computed by hand. The five refusals include `Double.NaN`, which would slip through the obvious range test. Two checks confirm that refused orders change nothing and fire nothing. The state machine of **Figure 11.11** is walked to its end and one step beyond. The read-only list is tested by trying to `clear()` it, which must throw `UnsupportedOperationException` and leave three orders. And the last check removes the counting listener and confirms that it hears nothing more: the event count stays at 6, from three orders and three advances. `expectRefused` takes the action to test as a `Runnable` lambda, so one helper serves all five refusals.

Listing 11.18 is the Python version with `unittest`.

Listing 11.18 `test_order_book.py`: unit tests of the front-desk model with `unittest`, using `subTest` for the five refusals.

```
"""test_order_book.py: unit tests for the front-desk model, with no window at all."""
import unittest

from order_book import OrderBook, OrderStatus, ServiceType


class OrderBookTest(unittest.TestCase):
    def setUp(self):
        self.book = OrderBook()
        self.events = 0
        self.book.add_listener(self.count_event)
        self.sari = self.book.add_order("  Sari Wulandari ", ServiceType.WASH_FOLD, 4.2,
False)
        self.budi = self.book.add_order("Budi", ServiceType.WASH_IRON, 2.6, True)
        self.citra = self.book.add_order("Citra", ServiceType.IRON_ONLY, 1.3, False)

    def count_event(self, book):
        self.events += 1

    def test_prices_follow_the_rules(self):
        self.assertEqual(self.sari.customer, "Sari Wulandari")
        self.assertEqual(self.sari.chargeable_kg(), 4.5)
        self.assertEqual(self.sari.price(), 31_500)
        self.assertEqual(self.budi.price(), 45_000)   # minimum 3 kg, then x 3/2
        self.assertEqual(self.citra.price(), 10_000)  # 2 kg minimum
        self.assertEqual(self.book.total_revenue(), 86_500)
        self.assertEqual(self.events, 3)

    def test_bad_orders_are_refused_and_change_nothing(self):
        bad = [("   ", ServiceType.WASH_FOLD, 3), ("Dewi", ServiceType.WASH_FOLD, 0),
               ("Dewi", ServiceType.WASH_FOLD, 50.5),
               ("Dewi", ServiceType.WASH_FOLD, float("nan")), ("Dewi", None, 3)]
        for customer, service, kg in bad:
            with self.subTest(customer=customer, service=service, kg=kg):
                with self.assertRaises(ValueError):
                    self.book.add_order(customer, service, kg, False)
        self.assertEqual(self.book.count(), 3)
        self.assertEqual(self.events, 3)
```

Listing 11.18 (Continued) *test_order_book.py: unit tests of the front-desk model with unittest, using subTest for the five refusals.*

```
    def test_status_moves_forward_then_stops(self):
        for _ in range(3):
            self.book.advance(2)
        self.assertIs(self.budi.status, OrderStatus.PICKED_UP)
        with self.assertRaises(RuntimeError) as caught:
            self.book.advance(2)
        self.assertEqual(str(caught.exception), "order #2 was already picked up")
        self.assertEqual(self.book.count_with(OrderStatus.RECEIVED), 2)

    def test_orders_are_a_read_only_snapshot(self):
        with self.assertRaises(AttributeError):
            self.book.orders().append(None)  # a tuple has no append

    def test_removed_listener_hears_nothing(self):
        self.book.remove_listener(self.count_event)
        self.book.advance(1)
        self.assertEqual(self.events, 3)


if __name__ == "__main__":
    unittest.main()
```

`python3 -m unittest -v test_order_book` reports:

```
test_bad_orders_are_refused_and_change_nothing
(test_order_book.OrderBookTest.test_bad_orders_are_refused_and_change_nothing) ... ok
test_orders_are_a_read_only_snapshot
(test_order_book.OrderBookTest.test_orders_are_a_read_only_snapshot) ... ok
test_prices_follow_the_rules (test_order_book.OrderBookTest.test_prices_follow_the_rules)
... ok
test_removed_listener_hears_nothing
(test_order_book.OrderBookTest.test_removed_listener_hears_nothing) ... ok
test_status_moves_forward_then_stops
(test_order_book.OrderBookTest.test_status_moves_forward_then_stops) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.003s

OK
```

Two `unittest` features appear here for the first time. `self.subTest(...)` inside a loop turns one test method into several reported cases: if the third bad order were accepted by mistake, the failure report would name exactly that `customer`, `service` and `kg`, and the loop would continue with the fourth. And `setUp` builds the same three orders before every test, so each test reads like a small story that starts from a known shop. The listener is a *bound method*, `self.count_event`, rather than a lambda, and that is not an accident: to unsubscribe a listener you must pass the *same* object to `remove_listener`, and `self.count_event` produces an equal bound method every time you write it, while two lambdas with the same body are two different objects. Section 11.13 returns to this trap.

11.11 Variations: MVP and MVVM

MVC has had many children, and two of them are so widespread that you should recognize them by name. **Figure 11.14** draws all three side by side.

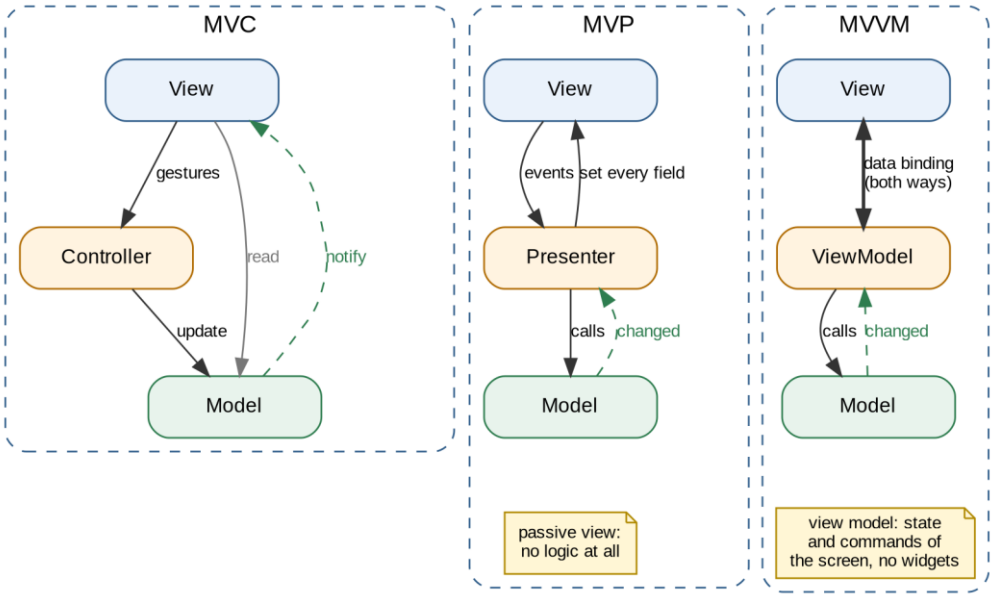


Figure 11.14 MVC and two of its descendants. In MVC the view observes the model and sends gestures to the controller. In MVP a passive view reports events to a presenter, which sets every field of the view. In MVVM the view is bound, in both directions, to a view model that holds the screen's state and commands.

Model–View–Presenter (MVP) was described by Taligent in 1996 [13] and made popular for testing by Michael Feathers's article "The Humble Dialog Box" [9]. In MVP the view is *passive*: it has no reference to the model at all, and it contains no logic, not even the logic of what to display. The **presenter** receives the view's events, talks to the model, and then *sets every field of the view* through an interface. If this sounds familiar, it should: our laundry desk is, strictly speaking, MVP. `DeskController` is a presenter; the view interfaces are the passive-view interfaces; and the text views of the self-test are exactly the "fake views" that MVP was invented to allow. The names in this chapter follow the more common word, *controller*, because the difference is one of degree: how much the view knows. In the keypad, the views observe the model directly — classic MVC; in the desk, they know nothing — MVP.

Model–View–ViewModel (MVVM) was introduced by Microsoft for its WPF toolkit in 2005 [14] and is now the standard structure of many phone and web frameworks. It adds a **view model**: an object that holds the *state of the screen* — the text in each field, whether a button is enabled, the message to show — as observable properties, and offers the screen's commands as methods. The view is connected to the view model by **data binding**: a declaration, made once, that a text field and a property always hold the same value, in both directions, kept in step by observers underneath. The view model never touches a widget, so it can be tested like a model. Tkinter has a small taste of this: an `Entry` created with `textvariable=var` is bound to a `StringVar` in both directions, and `var.trace_add("write", callback)` lets code observe it. Exercise 11.10 builds a miniature MVVM with a hand-written observable property, and shows the form computing a price preview and enabling the **Add order** button as the user types.

For the programs in this book the choice matters less than what the three share: the rules live in a model that knows nothing about the screen, and the logic is testable without a window. Martin Fowler's essay *GUI Architectures* [15] is the best short history of how these variants arose and what each was trying to fix.

11.12 Profitina in Practice

The laundry is our workshop; Profitina's real product shows the same ideas on a larger scale. Profitina is an AI visibility platform: it scans a brand's website and measures how ready the site is to be cited by AI answer engines, the discipline called generative engine optimization (GEO), and then helps fix what it finds. Its web application is written in Python with the FastAPI framework, and its user interface runs in the browser.

The clearest example of MVC in that product is the **dashboard**. When a user signs in, the dashboard shows, on one screen, notifications of replies, likes, reposts and direct messages, the status of feedback the user has sent, PROFIT purchases, the channels the user follows with counters of new posts, and the user's rank and activity. Each of these panels is a *view*, and each is a view over a different *model*: the community's posts, the feedback pipeline, the PROFIT ledger, the channel subscriptions. The dashboard owns none of that data. It is assembled by one call, `GET /api/me/dashboard`, which gathers what the views need from the models and returns it; each item on the screen is clickable and leads to the section that owns it. That is Equation (11.1) at work: the dashboard is a function of the models' state, and a dashboard that stored its own copy of the counts would, sooner or later, show a count that disagrees with the channel it summarizes.

The Observer pattern is equally visible. The real-time community feed uses **Server-Sent Events**: the browser subscribes once to a stream, and the server pushes a message whenever a new post appears — a subject notifying its listeners across a network. Posting is *optimistic*: the new post appears in the view at once, and if sending fails the text is restored — a deliberate, carefully undone exception to "the model first".

And the testing strategy of Sections 11.8 and 11.10 is Profitina's release rule. Its full automated suite — well over a thousand tests, most of them checking models and request handlers without any browser, plus guards for the JavaScript interface — must pass before a release is allowed. That is only practical because the logic lives where a fast test can reach it. As everywhere in Profitina, the scores and suggestions such views display are advice: the platform explains what it measured, and the user decides what to do.

11.13 Common Mistakes, and How to Avoid Them

1. Business rules in a button listener. The price, the minimum weight or the validity check computed inside `actionPerformed` or a Tkinter `command`. *Avoid it:* the listener's body should be one line that calls the controller; the rule belongs in the model, where a unit test can reach it (Section 11.2).

2. A model that imports the GUI toolkit. One `import javax.swing.JOptionPane` in the model "just to show an error" ties the model to a screen forever. *Avoid it:* the model throws an exception or returns a result; the controller decides how to show it. Check by searching the model's file for `swing`, `awt` or `tkinter`.

3. Forgetting to draw the first picture. Observers are told about *changes*; at start-up nothing has changed, so the view stays empty until the first click. *Avoid it:* the controller (or the view) calls its refresh method once after wiring, as every controller of this chapter does.

4. Notifying before validating, or notifying once per field. A model that fires an event and then throws leaves views showing a value that never existed; a model that fires three events for one operation triples the redrawing (Equation (11.2)). *Avoid it:* validate, change, notify once, return.

5. Changing the listener list while notifying. A listener that unsubscribes itself during notification makes a plain `for` loop skip the next listener in both languages, silently (Exercise 11.3). *Avoid it:* iterate over a copy of the list (`List.copyOf`, `list(...)`), or use `PropertyChangeSupport`, which does so for you.

6. Subscribing a lambda you cannot unsubscribe. `model.add_listener(lambda b: ...)` followed later by `model.remove_listener(lambda b: ...)` fails, because the two lambdas are different objects: Python raises `ValueError`, and Java's `remove` quietly finds nothing to remove. A view that is closed but never unsubscribed stays alive, and keeps redrawing, as long as the model lives — the *lapsed listener* problem. *Avoid it:* keep a reference to the listener you registered, or subscribe a bound method, and unsubscribe when the view closes.

7. Creating Swing components outside the event dispatch thread, or on a machine without a display. Swing is single-threaded: build and change widgets only inside `SwingUtilities.invokeLater` [8]. On a computer without a display, the first `JFrame` fails; running `java CounterMvc` on the machine that built this book produced:

```
Exception in thread "AWT-EventQueue-0" java.awt.HeadlessException:
No X11 DISPLAY variable was set,
or no headful library support was found,
but this program performed an operation which requires it.
```

Avoid it: keep the logic out of the window and test it with `--selftest` and unit tests, which never create one. The Python equivalent on a machine without Tkinter is `ModuleNotFoundError: No module named 'tkinter'`, avoided in the same way — and by importing `tkinter` only in the view.

8. A lambda in a loop that captures the loop variable (Python). `command=lambda: controller.press(key)` inside `for key in "123..."` makes every button press the *last* key. *Avoid it:* capture the current value with a default argument, `lambda k=key: controller.press(k)` (Section 11.7.5).

9. Believing the self-test proves the window is right. The self-test exercises the model and the controller; it cannot see that a button has disappeared off the edge of a panel (Section 11.9.5) or that a label says *vwxy* instead of *wxyz*. *Avoid it:* keep the view free of logic, generate its labels from the model's own tables, and look at the window after every change to it.

10. Calling one class `View` and letting it do the controller's work. A class named `View` that implements `ActionListener` and calls the model in `actionPerformed` is a view *and* a controller, whatever its name. *Avoid it:* ask of every method "does it draw, decide, or translate?", and move it to the class whose job that is.

11.14 Chapter Summary

- A program that puts widgets, business rules and data in one class cannot show the data twice, change the rules safely, be tested without a screen, or move to a new interface. **MVC** splits it into a **model** (data and rules, no GUI imports), **views** (presentation, no rules) and a **controller** (translation of gestures into model operations).
- Ideally a view is a function of the model's state, $V = R(S)$ (Equation (11.1)); then several views of one model can never disagree (**Figure 11.4**), and the model can be tested alone.
- The **Observer** pattern lets the model notify views without knowing them: a subject keeps a list of observers and calls each on every change. Java's `Observable` is deprecated; use `PropertyChangeSupport`, which fires named events with old and new values and skips non-

changes. In Python, a list of callables is enough. Notification is synchronous and costs $m \cdot k$ calls (Equation (11.2)), so notify **once per logical change**; iterate over a copy of the listener list.

- Observers can be **pushed** the new value or **pull** it from the model; pulling suits large models with many different views.
- The controller can let views observe the model directly (the keypad) or act as a **mediator** between them (the laundry desk, **Figure 11.5**), which is also called MVP.
- Give each view an **interface** (Java) or **protocol** (Python); then a text view can replace the window in a `--selftest` mode that runs the model and controller headless, and **unit tests** can check every rule of the model in milliseconds.
- Model methods follow **validate, change, notify, return**; the controller turns text into values, and the model judges the values.
- **MVP** makes the view passive and the presenter responsible for every field; **MVVM** binds the view to a view model of observable properties. All three keep the rules out of the window.

11.15 Exercises

Exercise 11.1 (which role?). For each of the following pieces of code from a laundry program, say whether it belongs in the model, a view or the controller, and give a one-sentence reason. (a) `price = Math.round(chargeableKg() * ratePerKg);`. (b) `table.setRowSelectionInterval(row, row);`. (c) `double kg = Double.parseDouble(weightField.getText().replace(',', '.'));`. (d) `if (status == OrderStatus.PICKED_UP) throw new IllegalStateException(...);`. (e) `summary.setText(count + " orders");`. (f) `if (key == '0') model.acceptWord();`. (g) `listeners.add(listener);`.

Exercise 11.2 (a second view). Add to the student program of Section 11.4 a second view that prints each student as one line of comma-separated values, roll number first (`10,Robert`). Let the controller hold any number of views and update all of them. Do not change the class `Student`.

Exercise 11.3 (a listener that leaves). Three listeners are registered in this order: `log`, which prints `log`; `once`, which prints `once` and then removes itself from the list; and `display`, which prints `display`. The subject notifies them with a plain `for` loop over the list itself (not a copy), twice. Predict the output in Java (with an `ArrayList` and an enhanced `for`) and in Python, then run both and explain what you see.

Exercise 11.4 (no change, no event). In **Listing 11.3**, `setLoads(2)` fired no event. (a) Which object made that decision, and where in the program's output can you see it? (b) What would happen if two models each observed the other and copied every new value into itself, and neither suppressed non-changes? (c) Why is it important that `setLoads` validates *before* calling `firePropertyChange`?

Exercise 11.5 (a rule for the counter). The laundry wants the counter of Section 11.6 to count loads in the washing machine, which holds at most 12. Change the program so that the count can never go below 0 or above 12. In which class does the rule go? What should the view show when *Up* is clicked at 12? Extend the self-test to prove the rule.

Exercise 11.6 (a label bug). In the original keypad program, key 9 was labeled *vwxy* while the lookup used *wxyz*. (a) Why could no unit test of the model have found this? (b) How does **Listing 11.7** make the bug impossible? (c) Which letter could a user of the original program not see on any key?

Exercise 11.7 (undo a word). Extend the keypad so that pressing `C` when no digits have been typed removes the last accepted word. Change the model, not the controller; add a unit test to both test programs; and make sure the model still fires exactly one event per real change.

Exercise 11.8 (the waiting-room screen). Write a new view of the laundry's `OrderBook` for the television in the waiting area: it shows only the orders that are *Ready*, as `#2 Budi`, and redraws only when that list of ready orders actually changes. Do not change `OrderBook.java` or `order_book.py`.

Exercise 11.9 (canceling an order). The laundry wants a **Cancel order** button that removes an order, but only while it is still *Received*. Describe every change you would make — in the model, the controller, the view interfaces, the Swing and Tkinter views, the text views and the tests — and write the new model method in both languages.

Exercise 11.10 (MVVM in miniature). Write a generic observable `Property` class with `get`, `set` (which notifies only on real changes), `observe` and `bind` (which also calls the observer once at once). Use it to write a view model for the order form, restricted to Wash & Fold, with input properties `customer`, `weightText` and `express` and output properties `preview` (for example `4.5 kg charged, Rp31,500`) and `canAdd`. Show the output as a user types a name, a weight, ticks *Express* and then types a bad weight.

Exercise 11.11 (a sequence diagram). Draw the sequence diagram for a click on **Advance status** when order 2 is selected and is *Washing*, in the style of **Figure 11.13**. Then draw it again for the case where order 2 has already been picked up. How many messages does the model send in each case?

Exercise 11.12 (the web version). Profitina Laundry opens a second branch and wants the front desk as a web page. Go through every class of Section 11.9 and say whether it can be kept unchanged, must be changed, or must be replaced. What single design decision made most of the answers "unchanged"?

11.16 Solutions to the Exercises

Solution 11.1. (a) *Model*: computing a price is a business rule. (b) *View*: which row is highlighted is presentation. (c) *Controller*: turning the text a user typed into a number is translation of a gesture into a value; the model should receive a `double`, never a text field. (d) *Model*: the order's life cycle is a business rule, enforced where the state lives. (e) *View*: formatting a count as words on a label is presentation. (f) *Controller*: that key 0 means "accept the word" is a fact about this keypad, not about messages. (g) *Model* (the subject side of Observer): the model keeps its listeners, though it knows nothing about what they are.

Solution 11.2. Listing 11.19 and **Listing 11.20** give each view a single method, so that "a view" is simply "something that can show a name and a roll number". The controller keeps a list of them.

Listing 11.19 `StudentTwoViews.java`: a controller with any number of views; the original `StudentView` and a new comma-separated view. It uses `Student` and `StudentView` from `StudentMvcDemo.java`.

```
import java.util.ArrayList;
import java.util.List;

public class StudentTwoViews {
    public static void main(String[] args) {
        Student model = new Student();
        model.setName("Robert");
        model.setRollNo("10");
        MultiViewController controller = new MultiViewController(model);
        controller.addView(new StudentView()::printStudentDetails);
        controller.addView(new CsvStudentView());
        controller.updateViews();
        controller.setStudentName("John");
        controller.updateViews();
    }
}

// Any view of a student: one method, so a lambda or a method reference will do.
interface StudentDisplay {
    void show(String name, String rollNo);
}

// A second view of the same model: one line, ready for a spreadsheet.
class CsvStudentView implements StudentDisplay {
    @Override
    public void show(String name, String rollNo) {
        System.out.println(rollNo + "," + name);
    }
}

class MultiViewController {
    private final Student model;
    private final List<StudentDisplay> views = new ArrayList<>();

    MultiViewController(Student model) {
        this.model = model;
    }

    void addView(StudentDisplay view) {
        views.add(view);
    }

    void setStudentName(String name) {
        model.setName(name);
    }
```

Listing 11.19 (Continued) *StudentTwoViews.java: a controller with any number of views; the original StudentView and a new comma-separated view. It uses Student and StudentView from StudentMvcDemo.java.*

```
    void updateViews() {
        for (StudentDisplay view : views) {
            view.show(model.getName(), model.getRollNo());
        }
    }
}
```

```
Student:
Name: Robert
Roll No: 10
10,Robert
Student:
Name: John
Roll No: 10
10,John
```

In Java the interface `StudentDisplay` has one abstract method, so it is a functional interface, and the original view can be adapted to it with the method reference `new StudentView()::printStudentDetails` — the old class is reused without modification. `Student` is untouched, as required.

Listing 11.20 `student_two_views.py`: the same two views in Python, importing the model and the first view from `student_mvc_demo.py`.

```
"""student_two_views.py: one student model, two views, one controller."""
from student_mvc_demo import Student, StudentView


class CsvStudentView:
    """A second view of the same model: one line, ready for a spreadsheet."""

    def show(self, name, roll_no):
        print(f"{roll_no},{name}")


class MultiViewController:
    def __init__(self, model):
        self._model = model
        self._views = []  # any callable taking (name, roll_no) is a view

    def add_view(self, view):
        self._views.append(view)

    def set_student_name(self, name):
        self._model.name = name

    def update_views(self):
        for view in self._views:
            view(self._model.name, self._model.roll_no)


def main():
    model = Student(name="Robert", roll_no="10")
    controller = MultiViewController(model)
    controller.add_view(StudentView().print_student_details)
    controller.add_view(CsvStudentView().show)
    controller.update_views()
    controller.set_student_name("John")
    controller.update_views()


if __name__ == "__main__":
    main()
```

```
Student:
Name: Robert
Roll No: 10
10,Robert
Student:
Name: John
Roll No: 10
10,John
```

In Python a view is any callable taking `(name, roll_no)`, so the controller stores bound methods: `StudentView().print_student_details` and `CsvStudentView().show`. Neither version changed the model: the model of a student does not care how many ways there are of printing it.

Solution 11.3. Both languages print the same, and neither complains:

```
round 1
log
once
round 2
log
display
```

In round 1, `display` is skipped. The Java enhanced `for` over an `ArrayList` uses an iterator that keeps a *cursor*, the index of the next element. After `log` (index 0) and `once` (index 1), the cursor is 2; `once` removed itself, so the list now has size 2, and the iterator's `hasNext()` tests `cursor != size`, finds `2 != 2` false, and ends the loop — without the `ConcurrentModificationException` that a removal elsewhere in the list would have caused. Python's list iterator also walks by index: after index 1 it tries index 2, which no longer exists in the two-element list, and stops. In round 2 `once` is gone and both remaining listeners run. The bug is silent, and it depends on the position of the listener that leaves — the worst kind. The cure is to notify from a copy: `for (Runnable l : List.copyOf(listeners))` or `for listener in list(listeners):`, as every subject of this chapter does.

Solution 11.4. (a) `PropertyChangeSupport.firePropertyChange` compares the event's old and new values and fires nothing when they are equal; in the output, the heading `setLoads(2)` is followed directly by the next heading, with no observer line between them. (b) An endless ping-pong: A notifies B, B copies the value and notifies A, A copies it back, and so on until the stack overflows (`StackOverflowError`, Python `RecursionError`). Suppressing non-changes stops it after one round. (c) If the model notified first and then threw, the observers would already have displayed -1, a value the model never actually accepted; validating first guarantees that observers only ever see legal states.

Solution 11.5. The rule belongs in the **model**, because it is a fact about washing machines, true however the count is displayed:

```
static final int MAX_LOADS = 12;

void setValue(int newValue) {
    if (newValue < 0 || newValue > MAX_LOADS) {
        return;                    // or throw, and let the controller explain
    }
    if (newValue == value) {
        return;
    }
    value = newValue;
    for (IntConsumer listener : List.copyOf(listeners)) {
        listener.accept(value);
    }
}
```

A click on *Up* at 12 now changes nothing and notifies nothing, so the view keeps showing 12. If the laundry wants a message ("the machine is full"), `setValue` should throw, and the controller should catch it and ask the view to show the message. The self-test gains a loop that clicks *Up* thirteen times from 0 and expects 12, and one that clicks *Down* at 0 and expects 0. Python needs the same checks in `set_value`.

Solution 11.6. (a) The labels were written by hand in the view, as string literals, and the model never sees them: the model's lookup table was correct, so every test of the model passed. A bug that lives only in the view is invisible to tests of the model — which is one reason to keep the view as small as possible. (b) `KeypadPanel` computes each label with `T9Dictionary.lettersOf(key)`, from the same `LETTERS` table that computes signatures, so the labels and the lookup cannot

disagree; there is only one copy of the fact. (c) The letter *z*: key 9's label showed *v*, *w*, *x* and *y*, and *v* already appears on key 8, so *z* appeared on no key at all, although typing 9 for *z* worked.

Solution 11.7. The change is in `deleteDigit`, which becomes "delete the last digit, or else the last accepted word":

```
void deleteDigit() {
    if (!signature.isEmpty()) {
        signature = signature.substring(0, signature.length() - 1);
        lookUp();
        fireChanged();
    } else if (!accepted.isEmpty()) {
        accepted.remove(accepted.size() - 1);
        fireChanged();
    }
}
```

The controller does not change, because `C` still means `deleteDigit`; only the meaning of the operation has grown (you might rename it `deleteBack`). In Python the new branch is `elif self._accepted: self._accepted.pop(); self._fire_changed()`. Two new checks are needed: one types `8430` (*the*), presses `C`, and expects an empty text; another counts events and expects exactly one for that `C` and none for a further `C` on an empty message.

One existing test must change, and it is instructive to see why. *One event per real change* types *hello*, accepts it, and then calls `deleteDigit` expecting "nothing to delete". Under the new rule that call removes *hello*: the count becomes 7 and the text empty. The failing test is doing its job — it reports that behavior has changed — and deciding that the new behavior is intended, and updating the test to say so, is the programmer's decision.

Solution 11.8. **Listing 11.21** and **Listing 11.22** add one class, a listener of the order book, and one line that subscribes it.

Listing 11.21 `PickupBoard.java`: a new view of the order book for the waiting-room screen, added without changing the model. Compile it with `OrderBook.java`.

```java
import java.util.ArrayList;
import java.util.List;

public class PickupBoard {
    public static void main(String[] args) {
        OrderBook book = new OrderBook();
        book.addListener(new PickupBoardView());  // the only new line: a new view
        System.out.println("> three orders arrive");
        book.addOrder("Sari Wulandari", ServiceType.WASH_FOLD, 4.2, false);
        book.addOrder("Budi", ServiceType.WASH_IRON, 2.6, true);
        book.addOrder("Citra", ServiceType.IRON_ONLY, 1.3, false);
        System.out.println("> Budi's order is washed, then ready");
        book.advance(2);
        book.advance(2);
        System.out.println("> Citra's order is washed, then ready");
        book.advance(3);
        book.advance(3);
        System.out.println("> Budi picks up");
        book.advance(2);
    }
}

// The waiting-room screen: it shows only orders that are ready, and redraws
// only when that list actually changes.
class PickupBoardView implements OrderBookListener {
    private String lastShown = "";

    @Override
    public void orderBookChanged(OrderBook book) {
        List<String> ready = new ArrayList<>();
        for (LaundryOrder order : book.getOrders()) {
            if (order.getStatus() == OrderStatus.READY) {
                ready.add("#" + order.getNumber() + " " + order.getCustomer());
            }
        }
        String board = ready.isEmpty() ? "(nothing yet)" : String.join("   ", ready);
        if (!board.equals(lastShown)) {
            System.out.println("  READY FOR PICKUP: " + board);
            lastShown = board;
        }
    }
}
```

```
> three orders arrive
  READY FOR PICKUP: (nothing yet)
> Budi's order is washed, then ready
  READY FOR PICKUP: #2 Budi
> Citra's order is washed, then ready
  READY FOR PICKUP: #2 Budi   #3 Citra
> Budi picks up
  READY FOR PICKUP: #3 Citra
```

The view computes its picture from the model's current state and redraws only when it differs from the last one: three new orders produced three notifications, but one drawing. This is the television of Section 11.2, and the model did not change.

Listing 11.22 `pickup_board.py`: the waiting-room view in Python; an object with `__call__` is a callable, so it can be registered as a listener.

```
"""pickup_board.py: a new view of the order book, added without touching the model."""
from order_book import OrderBook, OrderStatus, ServiceType


class PickupBoardView:
    """The waiting-room screen: it shows only orders that are ready, and
    redraws only when that list actually changes."""

    def __init__(self):
        self._last_shown = ""

    def __call__(self, book):
        ready = [f"#{o.number} {o.customer}" for o in book.orders()
                 if o.status is OrderStatus.READY]
        board = "   ".join(ready) if ready else "(nothing yet)"
        if board != self._last_shown:
            print("  READY FOR PICKUP: " + board)
            self._last_shown = board


def main():
    book = OrderBook()
    book.add_listener(PickupBoardView())  # the only new line: a new view
    print("> three orders arrive")
    book.add_order("Sari Wulandari", ServiceType.WASH_FOLD, 4.2, False)
    book.add_order("Budi", ServiceType.WASH_IRON, 2.6, True)
    book.add_order("Citra", ServiceType.IRON_ONLY, 1.3, False)
    print("> Budi's order is washed, then ready")
    book.advance(2)
    book.advance(2)
    print("> Citra's order is washed, then ready")
    book.advance(3)
    book.advance(3)
    print("> Budi picks up")
    book.advance(2)


if __name__ == "__main__":
    main()
```

```
> three orders arrive
  READY FOR PICKUP: (nothing yet)
> Budi's order is washed, then ready
  READY FOR PICKUP: #2 Budi
> Citra's order is washed, then ready
  READY FOR PICKUP: #2 Budi   #3 Citra
> Budi picks up
  READY FOR PICKUP: #3 Citra
```

Solution 11.9. *Model:* a new status or, more simply, removal from the list. In Java:

```
public void cancel(int number) {
    LaundryOrder order = find(number);
    if (order.getStatus() != OrderStatus.RECEIVED) {
        throw new IllegalStateException("order #" + number + " is already being washed");
    }
    orders.remove(order);
    fireChanged();
}
```

and in Python:

```
def cancel(self, number):
    order = self.find(number)
    if order.status is not OrderStatus.RECEIVED:
        raise RuntimeError(f"order #{number} is already being washed")
    self._orders.remove(order)
    self._fire_changed()
```

View interfaces: `OrderListView` gains `onCancel(Runnable)` / `on_cancel(action)`. *Controller:* a method `cancelSelected()`, registered in the constructor, shaped exactly like `advanceSelected()`. *Swing and Tkinter views:* one more button next to **Advance status**, whose listener or command runs the registered action. *Text views:* a `clickCancel()` method. *Tests:* cancel a received order (count and revenue drop, one event), refuse to cancel an order that is washing (nothing changes, no event), and extend the self-test. The summary view needs no change at all: it recomputes its numbers from the model.

Solution 11.10. **Listing 11.23** and **Listing 11.24** write the property and the view model. The "view" is two lambdas bound to the output properties; in a real program they would set a label's text and a button's enabled state.

Listing 11.23 `MiniBinding.java`: a generic observable `Property` and a view model of the order form. Compile it with `OrderBook.java`.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.Objects;
import java.util.function.Consumer;

public class MiniBinding {
    public static void main(String[] args) {
        OrderFormViewModel vm = new OrderFormViewModel();
        // The "view": two widgets bound to view-model properties.
        vm.preview.bind(text -> System.out.println("  preview label: " + text));
        vm.canAdd.bind(ok -> System.out.println("  Add order enabled: " + ok));
        System.out.println("> type customer \"Sari\"");
        vm.customer.set("Sari");
        System.out.println("> type weight \"4.2\"");
        vm.weightText.set("4.2");
        System.out.println("> tick Express");
        vm.express.set(true);
        System.out.println("> type weight \"4,2x\"");
        vm.weightText.set("4,2x");
    }
}

// A value that tells its observers whenever it really changes.
class Property<T> {
    private final List<Consumer<T>> observers = new ArrayList<>();
    private T value;

    Property(T initial) {
        value = initial;
    }

    T get() {
        return value;
    }

    void set(T newValue) {
        if (!Objects.equals(value, newValue)) {
            value = newValue;
            for (Consumer<T> observer : List.copyOf(observers)) {
                observer.accept(value);
            }
        }
    }
```

Listing 11.23 (Continued) *MiniBinding.java: a generic observable Property and a view model of the order form. Compile it with OrderBook.java.*

```
    void observe(Consumer<T> observer) {
        observers.add(observer);
    }

    // Like observe, but also shows the current value at once.
    void bind(Consumer<T> observer) {
        observe(observer);
        observer.accept(value);
    }
}

// View model of the order form (Wash & Fold only, to keep it short):
// input properties in, display properties out, no widgets anywhere.
class OrderFormViewModel {
    final Property<String> customer = new Property<>("");
    final Property<String> weightText = new Property<>("");
    final Property<Boolean> express = new Property<>(false);
    final Property<String> preview = new Property<>("enter a weight");
    final Property<Boolean> canAdd = new Property<>(false);

    OrderFormViewModel() {
        customer.observe(v -> recompute());
        weightText.observe(v -> recompute());
        express.observe(v -> recompute());
    }

    private void recompute() {
        double kg;
        try {
            kg = Double.parseDouble(weightText.get().strip().replace(',', '.'));
        } catch (NumberFormatException e) {
            kg = Double.NaN;
        }
        if (!(kg > 0 && kg <= OrderBook.MAX_KG)) {
            preview.set("enter a weight");
            canAdd.set(false);
            return;
        }
        LaundryOrder draft = new LaundryOrder(0, customer.get(), ServiceType.WASH_FOLD,
                kg, express.get());
        preview.set(String.format(Locale.US, "%.1f kg charged, %s",
                draft.chargeableKg(), OrderBook.rupiah(draft.price())));
        canAdd.set(!customer.get().isBlank());
    }
}
```

```
  preview label: enter a weight
  Add order enabled: false
> type customer "Sari"
> type weight "4.2"
  preview label: 4.5 kg charged, Rp31,500
  Add order enabled: true
> tick Express
  preview label: 4.5 kg charged, Rp47,250
> type weight "4,2x"
  preview label: enter a weight
  Add order enabled: false
```

`bind` displays the initial values at once. Typing the name changes no output property, so nothing is redrawn. The weight 4.2 produces a preview and enables the button; ticking *Express* changes only the preview (Rp47,250 is $\lfloor 3 \times 31500/2 \rfloor$ by Equation (11.6)), so only the preview label is redrawn; and a weight of `4,2x` disables the button again. The view model reuses the model's

pricing rules through `LaundryOrder` rather than repeating them — a view model that computed prices itself would be the tangle of Section 11.2 in a new place.

Listing 11.24 `mini_binding.py`: the same property and view model in Python.

```
"""mini_binding.py: MVVM in miniature, with observable properties."""
from order_book import MAX_KG, LaundryOrder, ServiceType, rupiah


class Property:
    """A value that tells its observers whenever it really changes."""

    def __init__(self, initial):
        self._value = initial
        self._observers = []

    def get(self):
        return self._value

    def set(self, new_value):
        if new_value != self._value:
            self._value = new_value
            for observer in list(self._observers):
                observer(new_value)

    def observe(self, observer):
        self._observers.append(observer)

    def bind(self, observer):
        """Like observe, but also shows the current value at once."""
        self.observe(observer)
        observer(self._value)


class OrderFormViewModel:
    """View model of the order form (Wash & Fold only, to keep it short):
    input properties in, display properties out, no widgets anywhere."""

    def __init__(self):
        self.customer = Property("")
        self.weight_text = Property("")
        self.express = Property(False)
        self.preview = Property("enter a weight")
        self.can_add = Property(False)
        for source in (self.customer, self.weight_text, self.express):
            source.observe(lambda _value: self._recompute())
```

Listing 11.24 (Continued) *mini_binding.py: the same property and view model in Python.*

```
    def _recompute(self):
        try:
            kg = float(self.weight_text.get().strip().replace(",", "."))
        except ValueError:
            kg = float("nan")
        if not (0 < kg <= MAX_KG):
            self.preview.set("enter a weight")
            self.can_add.set(False)
            return
        draft = LaundryOrder(0, self.customer.get(), ServiceType.WASH_FOLD,
                             kg, self.express.get())
        self.preview.set(f"{draft.chargeable_kg():.1f} kg charged,
{rupiah(draft.price())}")
        self.can_add.set(self.customer.get().strip() != "")


def main():
    vm = OrderFormViewModel()
    # The "view": two widgets bound to view-model properties.
    vm.preview.bind(lambda text: print("  preview label: " + text))
    vm.can_add.bind(lambda ok: print("  Add order enabled: " + str(ok).lower()))
    print('> type customer "Sari"')
    vm.customer.set("Sari")
    print('> type weight "4.2"')
    vm.weight_text.set("4.2")
    print("> tick Express")
    vm.express.set(True)
    print('> type weight "4,2x"')
    vm.weight_text.set("4,2x")


if __name__ == "__main__":
    main()
```

```
  preview label: enter a weight
  Add order enabled: false
> type customer "Sari"
> type weight "4.2"
  preview label: 4.5 kg charged, Rp31,500
  Add order enabled: true
> tick Express
  preview label: 4.5 kg charged, Rp47,250
> type weight "4,2x"
  preview label: enter a weight
  Add order enabled: false
```

Solution 11.11. With order 2 *Washing*: 1 user → table view: click *Advance status*; 2 view → controller: `advanceSelected()`; 3 controller → list view: `selectedOrderNumber()` returns 2; 4 controller → model: `advance(2)`; 5 model → model: find, check, set status *Ready*; 6 model → controller: `orderBookChanged(book)`; 7 controller → list view: `showOrders(orders)`; 8 controller → summary: `showSummary(...)`; 9 model → controller: return; 10 controller → model: `find(2)` to read the new status; 11 controller → form: `showMessage("Order #2 is now Ready")`. The model sends one message to another object (6), plus the return. When order 2 has been picked up, message 5 ends by throwing `IllegalStateException`; messages 6 to 10 do not happen; the exception travels back to the controller, which catches it and sends `showMessage("Not changed: order #2 was already picked up")`. The model then sends *no* message at all — it only throws — which is exactly what "a refused operation changes nothing and notifies nobody" means.

Solution 11.12. *Unchanged:* `OrderBook`, `LaundryOrder`, `ServiceType`, `OrderStatus` and `OrderBookListener` — the whole model — and its unit tests. *Replaced:* the views (HTML pages or

a JavaScript front end instead of Swing or Tkinter). *Changed:* the controller, which receives HTTP requests instead of clicks but still turns text into values and calls the same model methods; if the web layer implements the view contracts, `DeskController` can even stay as it is. The decision that made the model reusable is the rule of **Table 11.1**: the model imports nothing from any user-interface toolkit.

References

[1] R. Eckstein, "Java SE application design with MVC," Oracle Technical Article, 2007. [Online]. Available: https://www.oracle.com/technical-resources/articles/javase/application-design-with-mvc.html

[2] T. Reenskaug, "Models–Views–Controllers," Technical note, Xerox PARC, Palo Alto, CA, USA, Dec. 1979. [Online]. Available: https://folk.universitetetioslo.no/trygver/themes/mvc/mvc-index.html

[3] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26–49, 1988.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

[5] A. Fowler, "A Swing architecture overview," Sun Microsystems / Oracle Technical Article. [Online]. Available: https://www.oracle.com/java/technologies/a-swing-architecture.html

[6] TutorialsPoint, "Design patterns – MVC pattern." [Online]. Available: https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm

[7] Oracle, *Java Platform, Standard Edition 21 API Specification*: `java.util.Observable` (deprecated since 9), `java.beans.PropertyChangeSupport`, `java.beans.PropertyChangeListener`. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/

[8] Oracle, "Concurrency in Swing," *The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/uiswing/concurrency/

[9] M. Feathers, "The humble dialog box," Object Mentor, 2002. [Online]. Available: https://martinfowler.com/articles/humble-dialog-box.pdf

[10] Python Software Foundation, *The Python Standard Library*, version 3.14: `tkinter`, `tkinter.ttk`, `typing.Protocol`, `unittest`. [Online]. Available: https://docs.python.org/3/library/

[11] D. L. Grover, M. T. King, and C. A. Kushler, "Reduced keyboard disambiguating computer," U.S. Patent 5 818 437, Oct. 6, 1998.

[12] The JUnit Team, *JUnit User Guide*, version 6.1.3 (Aug. 2026), and "Release notes: 6.0.0" (Java 17 baseline). [Online]. Available: https://docs.junit.org/current/

[13] M. Potel, "MVP: Model-View-Presenter, the Taligent programming model for C++ and Java," Taligent, Inc., Cupertino, CA, USA, 1996.

[14] J. Gossman, "Introduction to Model/View/ViewModel pattern for building WPF apps," Microsoft developer blog, Oct. 2005.

[15] M. Fowler, "GUI architectures," 2006. [Online]. Available: https://martinfowler.com/eaaDev/uiArchs.html

Chapter 12

Evaluation and Discussion III: Predictive Text with Lists, Maps, Trees, and a Keypad

This chapter adds no new language features. It works sixteen problems instead, each solved from a blank page in Java and in Python. The first ten form one project: the *predictive text* of the old mobile phones, where you press each key once and the phone guesses the word. We build its dictionary four times, reading the file on every question, keeping a sorted list, keeping a map, and keeping a tree that finds words while they are still being typed; we time all four; and we put a keypad on the screen with the Model–View–Controller pattern of Chapter 11. Six more problems are new, one for each idea of Chapters 9 to 11 that deserves a second look: choosing a collection by measuring it, a bounded generic class, a comparator chain, a counter whose model is tested without a window, two observer bugs, and buttons made in a loop. Every problem follows the route of Chapters 4 and 8: restate it, design it, solve it in both languages, run the tests, explain the code line by line, study a tempting wrong solution, and check yourself with a variant. By the end you will choose data structures by the operations they must support, prove the choice with measurements, and keep user interfaces thin enough to test without them.

12.1 Where We Left Off

Chapters 9, 10 and 11 gave us three tools, and this chapter uses them together.

From **Chapter 9** we have *collections* and *generics*. A collection is an object that holds other objects: a *list* (an ordered sequence that may contain duplicates: `ArrayList` in Java, `list` in Python), a *set* (no duplicates: `HashSet` and `TreeSet`, Python's `set`) or a *map* (from keys to values: `HashMap` and `TreeMap`, Python's `dict`). Generics let one class serve many element types while the compiler still checks them: `ArrayList<String>` holds only strings. We also met `Comparable` and `Comparator`, Java's two ways of ordering objects, and the `key=` argument that does the same in Python.

From **Chapter 10** we have *graphical user interfaces* and *events*. A Swing or Tkinter program builds a window from components and waits; when the user clicks, the toolkit calls the *listener* (Java) or *callback* (Python) attached to the button, often written as a lambda or an inner class.

From **Chapter 11** we have *Model–View–Controller*. The *model* holds the data and the rules and knows nothing about windows, the *view* shows the model, and the *controller* turns user actions into calls on the model. When the model changes, it tells every registered view: the *observer* pattern [1].

The chapter has four layers: a method (Section 12.2); the predictive-text project (Sections 12.3 to 12.7, Problems 12.1 to 12.10); six new problems set in the Profitina Laundry shop that runs through this book (Sections 12.8 and 12.9, Problems 12.11 to 12.16); and the usual closing sections. Each problem has the same bold labels, so that you can find your way back: **The problem**, **In our own words**, **Design**, **The Java solution**, **The Python solution**, **Line by line**, **A**

tempting wrong solution, and **Check yourself**. As in Chapter 8, every Java program tests itself with a small `check` helper that prints `PASS` and a name, and every Python program uses `unittest` [2]. Programs that measure time print different numbers on every computer; the numbers here are real ones from the machine the book was built on, and what matters is their *shape*: which column grows, and how fast.

> **HOW TO USE THIS CHAPTER.** Read a problem, cover the rest of the page, and spend ten minutes on it. For the dictionary problems, first write down *which operations* the structure must support and *how often*; for the GUI problems, *what the model must remember*. Then read on and compare.

12.2 How to Approach a Collections-and-Interface Problem

Problems about collections look like problems about code, but they are problems about *operations*. "Store the dictionary" is not a specification; "given a signature, find its words a thousand times a second, while new words arrive now and then" is. Once the operations and their frequencies are on paper, the structure almost chooses itself, and the user interface becomes a thin layer that calls those operations. **Table 12.1** condenses the method into six steps.

> **DEFINITION.** The *cost* of an operation is the number of basic steps it takes, as a function of the size n of the data. $O(f(n))$ ("order of $f(n)$") denotes a cost that grows at most like $f(n)$. $O(1)$ does not depend on n; $O(\log n)$ grows by one step when n doubles; $O(n)$ doubles when n doubles.

Step 1 — List the operations and how often each happens. *Add a word*: now and then. *Find the words of a signature*: at every key press. *Print everything in order*: once a day. The frequent operations decide the design; the rare ones only have to be possible.

Step 2 — Match each frequent operation to a structure. "Is this present?" and "what belongs to this key?" are the job of a hash set or hash map, $O(1)$ on average. "Everything in order" or "everything between two values" is the job of a sorted structure (a sorted list with binary search, `TreeSet`, `TreeMap`). "Everything that starts with this prefix" is the job of a tree whose paths spell the keys. A plain list is right only when you mostly append and walk through everything.

Step 3 — Hide the choice behind an interface. Code that *uses* the structure is written once against the interface, and changing the structure changes one line.

Step 4 — Put the state in a model that knows nothing about windows. Decide what the program must *remember* between two clicks, and put exactly that in a class with ordinary methods.

Step 5 — Test the model without a window. Ordinary methods get ordinary tests; a `--selftest` switch lets a GUI program click its own buttons on a machine without a screen.

Step 6 — Measure, with the same data for every candidate. A cost formula predicts the *shape* of the running time; a stopwatch confirms it.

Table 12.1 Six steps for a problem that combines collections, interfaces and a user interface. Steps 1 to 3 choose the data structure, Steps 4 and 5 shape the GUI, and Step 6 checks the choice.

Step	Write on paper	Question it answers
1	Operations and their frequencies	What must the data do, and how often?
2	A structure for each frequent operation	Which collection does that cheaply?
3	An interface with those operations	How do I keep the choice replaceable?
4	The model's fields and methods	What must the program remember between events?
5	Tests for the model; a `--selftest` for the GUI	How do I test without clicking?
6	A timing table with the same data for all	Does the prediction hold?

Two warnings belong to Step 6. The Java virtual machine compiles frequently used code into machine code while the program runs (*just-in-time compilation*), so early runs are slower than later ones; our timing programs repeat each measurement and keep the best. And a tiny measurement is mostly noise: make the data large and the questions many.

12.3 The Predictive-Text Problem

12.3.1 Keys, letters and signatures

Before phones had full keyboards they had twelve keys, eight of them carrying letters, laid out as in **Figure 12.1**: key 2 carries a, b and c; key 7 carries p, q, r and s; key 9 carries w, x, y and z; key 0 is the space; keys 1, `*` and `#` carry no letters.



Figure 12.1 The telephone keypad. Eight keys carry letters; keys 7 and 9 (shaded) carry four letters each, the other six carry three; key 0 is the space.

In the old system, a letter was typed by pressing its key once to four times: "hello" took thirteen presses, 4, 4, 3, 3, 5, 5, 5, pause, 5, 5, 5, 6, 6, 6. *Predictive text*, sold as T9 [3], asks for each key only once: you type 4, 3, 5, 5, 6, and the phone looks up the words that those five presses could mean. The row of digits is the word's *signature*. If $\kappa(c)$ is the key that carries the letter c, the signature of a word $w = c_1 c_2 \ldots c_m$ is

$$\mathrm{sig}(c_1 c_2 \ldots c_m) = \kappa(c_1)\,\kappa(c_2)\,\ldots\,\kappa(c_m) \tag{12.1}$$

a string of m digits, not a product. By Equation (12.1), sig(hello) = 43556 and sig(home) = 4663. The function cannot be undone uniquely, because words share signatures: sig(home) = sig(good) = sig(gone) = 4663. That is why the phone needs a dictionary and offers candidates. In our program, `*` moves to the next candidate, `0` accepts the word and types a space, and `#` deletes the last letter.

12.3.2 The dictionary file

A dictionary is a text file with one word per line. For the examples we use `words.txt`, 99 lines, in the programs' folder. Its beginning and end look like this:

```
a
act
bad
bat
...
worm
don't
e-mail
U.S.
3rd

café
  soap
London
```

The file is deliberately a little dirty, as real word lists are: an apostrophe, a hyphen, an abbreviation, a digit, an empty line, an accented letter, spaces around a word, capitalized duplicates (`Hello`, `Home`) and a proper noun. Every program follows one rule: trim the line, turn it into lower case, and use it only if it is *valid*, that is, not empty and made of the letters a to z only. So `Home` becomes `home` (and must not appear twice in any answer), `soap` becomes `soap`, `London` becomes `london`, and the other six odd lines are skipped. 93 lines survive.

For the timing experiments we need a much larger dictionary, so we generate one: `MakeDictionary` and `make_dictionary.py` write 200,000 invented but pronounceable words (`ulolifasu`, `gaweju`, …) to `big_words.txt`, identically in both languages (Section 12.5.3).

12.3.3 Packages and how to run the programs

The predictive-text classes live in the Java *package* `predictive`, the GUI classes in `predictivegui`. A package is a named group of classes; its name heads every file (`package predictive;`) and must match the file's folder. From the chapter's `java` folder, which holds `words.txt` and the two package folders:

```
javac predictive/*.java predictivegui/*.java *.java
java predictive.Sigs2WordsList 4663 43556
java predictivegui.PredictiveGUI
```

The compiler takes file *paths* with slashes; `java` takes a class *name* with dots, typed from the folder *above* `predictive`. Python's folders `predictive` and `predictivegui` are packages too (each has an `__init__.py`), and a module inside one is run with `python -m predictive.sigs2words_list 4663 43556` from the folder above. Every program reads the dictionary named by the environment variable `PREDICTIVE_DICT`, or `words.txt` when it is not set: on Linux and macOS, `PREDICTIVE_DICT=big_words.txt java ...` sets it for one command; on Windows PowerShell, `$env:PREDICTIVE_DICT="big_words.txt"`. The folder's `README.md` gives every command for Windows, macOS and Linux, in the terminal and in VS Code.

12.3.4 The whole design at a glance

Figure 12.2 shows the classes we are about to write. The interface `Dictionary` has one method, `signatureToWords`. Three classes implement it with different collections inside: `DictionaryListImpl` (a sorted `ArrayList` of `WordSig` pairs), `DictionaryMapImpl` (a `HashMap` from signatures to sets of words) and `DictionaryTreeImpl` (a tree with up to eight children per node). All use the static helpers of `PredictivePrototype`, and the GUI's `DictionaryModel` uses *any* `Dictionary`.

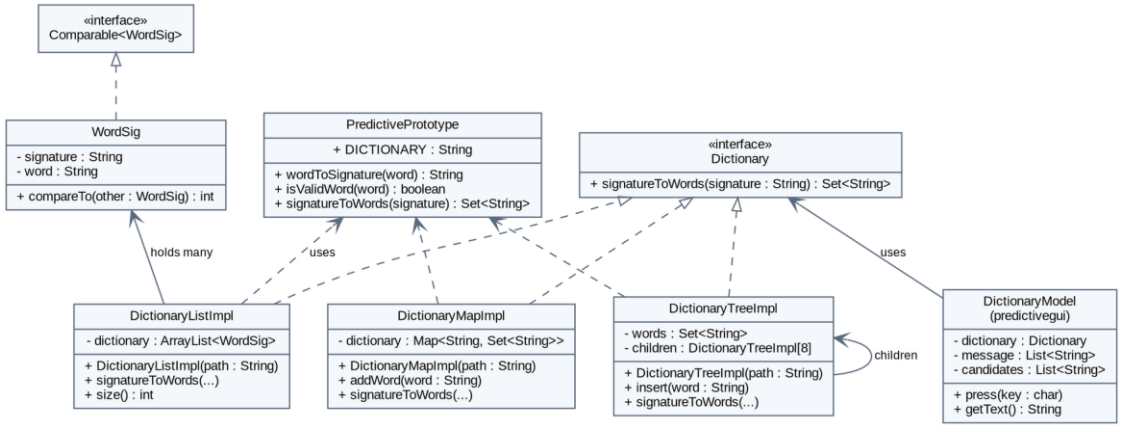


Figure 12.2 UML class diagram of the predictive-text project. Dashed lines with hollow triangles mean "implements"; solid arrows mean "has" or "uses"; `DictionaryTreeImpl` refers to objects of its own class.

Table 12.2 maps the ten problems of the project to four parts.

Table 12.2 The four parts of the predictive-text project and the chapters whose ideas each part uses.

Part	Problems	What is built	Chapter it exercises
One: prototypes	12.1–12.3	signatures, a file-scanning lookup, two command-line programs	9 (sets), 5 (files)
Two: a sorted list	12.4–12.6	`WordSig`, binary search, the `Dictionary` interface, first timings	9 (lists, `Comparable`), 6 (interfaces)
Three: a map and a tree	12.7–12.9	a multi-valued map, a prefix tree, timings of all four	9 (maps), 8 (recursion)
Four: a keypad	12.10	model, view and controller in Swing and Tkinter	10, 11

12.4 Part One: Prototypes

A *prototype* is a first version that may be slow, as long as it is simple and obviously correct. It pins down the right answers, and later it is the yardstick against which faster versions are checked and timed.

12.4.1 Problem 12.1 — From a word to its signature

> **PROBLEM.** In a class `PredictivePrototype` of the package `predictive`, write the method `public static String wordToSignature(String word)`. It takes a word and returns its numeric signature; for example, `"home"` gives `"4663"`. If the word contains a character that is not a letter, that character becomes a space in the signature. Build the result character by character in a `StringBuffer` rather than a `String`, and explain in a comment why this is more efficient.

In our own words. Walk through the word from left to right; for each character, ask which key carries it and append that key's digit. A character that no key carries (an apostrophe, a hyphen, a digit, an accented letter) contributes a space, so the signature keeps one position per character. Capitals are letters, so `"Home"` also gives `"4663"`.

Design. The heart of the method is the map κ of Equation (12.1) from a letter to its key. Java's *switch expression* writes it as a chain of cases; Python's dictionary writes it as a table built once. **Table 12.3** traces the loop by hand.

Table 12.3 The loop of `wordToSignature("Home")`, one character per step. The buffer grows by one digit at every step and is turned into a `String` once, at the end.

Step i	Character	Lower case	Key κ	Buffer after the step
0	`H`	`h`	`4`	`4`
1	`o`	`o`	`6`	`46`
2	`m`	`m`	`6`	`466`
3	`e`	`e`	`3`	`4663`

Why a `StringBuffer`? A Java `String` is *immutable*: `s = s + '6'` does not extend `s` but creates a new string, copies all the old characters into it and adds one. For a word of n letters the loop therefore copies

$$1 + 2 + \cdots + n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} \tag{12.2}$$

characters, about $n^2/2$: the work grows with the *square* of the length. A `StringBuffer` keeps an array with room to spare and appends in place, doubling the array on the rare occasions it fills up, so the total work stays proportional to n. (`StringBuilder` is the same class without the locking that makes `StringBuffer` safe for several threads, the subject of Chapter 14; for a local variable either will do.)

The Java solution. **Listing 12.1** is the complete class: this problem's method, the helper `isValidWord`, and the lookup `signatureToWords` of Problem 12.2.

Listing 12.1 `PredictivePrototype.java`: the signature of a word, the validity test and the file-scanning lookup, all as static methods of one utility class.

```java
// PredictivePrototype.java - Problems 12.1 and 12.2: a word's keypad signature,
// and a deliberately simple lookup that reads the dictionary file every time.
package predictive;

import java.io.File;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

public class PredictivePrototype {

    /** The dictionary file: the environment variable PREDICTIVE_DICT, else words.txt. */
    public static final String DICTIONARY =
            System.getenv().getOrDefault("PREDICTIVE_DICT", "words.txt");

    private PredictivePrototype() {             // a utility class: no objects
    }

    /**
     * Returns the keypad signature of a word, for example "home" gives "4663".
     * A character that is not a letter from a to z becomes a space.
     * The result is accumulated in a StringBuffer, which grows in place. Writing
     * result = result + digit on a String would copy every character collected
     * so far at each step: about n*n/2 copies for a word of n letters instead of n.
     */
    public static String wordToSignature(String word) {
        StringBuffer result = new StringBuffer(word.length());
        for (int i = 0; i < word.length(); i++) {
            result.append(keyOf(Character.toLowerCase(word.charAt(i))));
        }
        return result.toString();
    }

    /** The key that carries a letter: a, b, c are on '2', ..., w, x, y, z on '9'. */
    static char keyOf(char letter) {
        return switch (letter) {
            case 'a', 'b', 'c' -> '2';
            case 'd', 'e', 'f' -> '3';
            case 'g', 'h', 'i' -> '4';
            case 'j', 'k', 'l' -> '5';
            case 'm', 'n', 'o' -> '6';
            case 'p', 'q', 'r', 's' -> '7';
            case 't', 'u', 'v' -> '8';
            case 'w', 'x', 'y', 'z' -> '9';
```

Listing 12.1 (Continued) *PredictivePrototype.java: the signature of a word, the validity test and the file-scanning lookup, all as static methods of one utility class.*

```
            default -> ' ';
        };
    }

    /** True when the word is not empty and has only the letters a to z. */
    public static boolean isValidWord(String word) {
        if (word.isEmpty()) {
            return false;
        }
        for (int i = 0; i < word.length(); i++) {
            char c = word.charAt(i);
            if (c < 'a' || c > 'z') {
                return false;
            }
        }
        return true;
    }

    /**
     * Returns every dictionary word whose signature is the given one, in lower case
     * and without duplicates (a Set cannot hold the same word twice).
     * Inefficient on purpose: every call opens the file and reads ALL of it, and
     * computes the signature of every word again, so one lookup costs time
     * proportional to the size of the whole dictionary, and a thousand lookups
     * read the same file a thousand times.
     */
    public static Set<String> signatureToWords(String signature) {
        Set<String> result = new HashSet<>();
        try (Scanner in = new Scanner(new File(DICTIONARY), StandardCharsets.UTF_8)) {
            while (in.hasNextLine()) {
                String word = in.nextLine().trim().toLowerCase();
                if (isValidWord(word) && wordToSignature(word).equals(signature)) {
                    result.add(word);
                }
            }
        } catch (IOException e) {
            throw new IllegalStateException("cannot open the dictionary " + DICTIONARY, e);
        }
        return result;
    }
}
```

A class with only static methods is a *utility class*, like `Math`, and its `private` constructor forbids pointless objects. `DICTIONARY` reads the environment variable `PREDICTIVE_DICT` from the map `System.getenv()`, with `"words.txt"` as the default. **Listing 12.2** tests the class, runs two tempting wrong solutions, and times building an n-digit signature with `String` and with `StringBuffer`.

Listing 12.2 `PrototypeCheck.java`: checks for Problems 12.1 and 12.2, two wrong variants, and a timing of `String` against `StringBuffer`.

```java
// PrototypeCheck.java - checks for Problems 12.1 and 12.2, with two tempting
// wrong solutions and a timing of String versus StringBuffer.
package predictive;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;

public class PrototypeCheck {

    /** Tempting but wrong: "three letters per key" arithmetic. */
    static String wordToSignatureWrong(String word) {
        StringBuffer result = new StringBuffer();
        for (char c : word.toCharArray()) {
            result.append((char) ('2' + (c - 'a') / 3));
        }
        return result.toString();
    }

    /** Tempting but wrong: a List, no lower-casing, no validity check. */
    static List<String> signatureToWordsWrong(String signature) throws Exception {
        List<String> result = new ArrayList<>();
        for (String line :
java.nio.file.Files.readAllLines(java.nio.file.Path.of("words.txt"))) {
            if (PredictivePrototype.wordToSignature(line).equals(signature)) {
                result.add(line);
            }
        }
        return result;
    }

    /** Builds a signature of n digits by String concatenation: n new Strings. */
    static String concatWithString(int n) {
        String s = "";
        for (int i = 0; i < n; i++) {
            s = s + '2';
        }
        return s;
    }
```

Listing 12.2 (Continued) *PrototypeCheck.java: checks for Problems 12.1 and 12.2, two wrong variants, and a timing of String against StringBuffer.*

```
    static String concatWithBuffer(int n) {
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < n; i++) {
            sb.append('2');
        }
        return sb.toString();
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) throws Exception {
        check("home", "4663", PredictivePrototype.wordToSignature("home"));
        check("hello", "43556", PredictivePrototype.wordToSignature("hello"));
        check("four-letter keys", "7777 9999", PredictivePrototype.wordToSignature("pqrs
wxyz"));
        check("upper case", "4663", PredictivePrototype.wordToSignature("Home"));
        check("apostrophe", "366 8", PredictivePrototype.wordToSignature("don't"));
        check("empty word", "", PredictivePrototype.wordToSignature(""));
        check("valid", true, PredictivePrototype.isValidWord("laundry"));
        check("invalid e-mail", false, PredictivePrototype.isValidWord("e-mail"));
        check("invalid empty", false, PredictivePrototype.isValidWord(""));

        Set<String> w = new TreeSet<>(PredictivePrototype.signatureToWords("4663"));
        check("4663", "[gone, good, goof, home, hone, hood, hoof]", w.toString());
        check("5477", "[kiss, lips]", new
TreeSet<>(PredictivePrototype.signatureToWords("5477")).toString());
        check("no match", "[]", PredictivePrototype.signatureToWords("22222").toString());

        System.out.println("wrong 'you'      : " + wordToSignatureWrong("you")
                + "   right: " + PredictivePrototype.wordToSignature("you"));
        System.out.println("wrong 4663 list  : " + signatureToWordsWrong("4663"));
```

Listing 12.2 (Continued) *PrototypeCheck.java: checks for Problems 12.1 and 12.2, two wrong variants, and a timing of String against StringBuffer.*

```
        for (int n : new int[] {20_000, 40_000, 80_000}) {
            long t0 = System.nanoTime();
            concatWithString(n);
            long t1 = System.nanoTime();
            concatWithBuffer(n);
            long t2 = System.nanoTime();
            System.out.printf("n = %6d   String: %7.2f ms   StringBuffer: %5.2f ms%n",
                    n, (t1 - t0) / 1e6, (t2 - t1) / 1e6);
        }
        System.out.println("All " + passed + " checks passed.");
    }
}
```

Run it from the `java` folder with `java predictive.PrototypeCheck`:

```
PASS home
PASS hello
PASS four-letter keys
PASS upper case
PASS apostrophe
PASS empty word
PASS valid
PASS invalid e-mail
PASS invalid empty
PASS 4663
PASS 5477
PASS no match
wrong 'you'      : :68   right: 968
wrong 4663 list  : [gone, good, goof, home, Home, hone, hood, hoof]
n =  20000   String:  120.26 ms   StringBuffer:  2.37 ms
n =  40000   String:  184.31 ms   StringBuffer:  1.71 ms
n =  80000   String:  837.10 ms   StringBuffer: 13.24 ms
All 12 checks passed.
```

Over the three sizes, n grows four times and the `String` time about seven times, in line with the square law of Equation (12.2) blurred by the virtual machine warming up; the `StringBuffer` column stays in the low milliseconds (its one larger value is noise from a busy machine). Your numbers will differ; the shape will not.

The Python solution. Python's `str` is immutable too and there is no `StringBuffer`; the idiom is to collect the pieces in a list, which grows in place, and join them once with `"".join(...)`. **Listing 12.3** does that, with a dictionary built once as the table κ, and **Listing 12.4** tests it.

Listing 12.3 `prototype.py`: the same three functions in Python. The table `_KEY_OF` is built once, when the module is first imported.

```
# prototype.py - Problems 12.1 and 12.2: a word's keypad signature,
# and a deliberately simple lookup that reads the dictionary file every time.
import os

DICTIONARY = os.environ.get("PREDICTIVE_DICT", "words.txt")

_KEY_OF = {}                                    # letter -> key, built once
for _digit, _letters in (("2", "abc"), ("3", "def"), ("4", "ghi"), ("5", "jkl"),
                         ("6", "mno"), ("7", "pqrs"), ("8", "tuv"), ("9", "wxyz")):
    for _letter in _letters:
        _KEY_OF[_letter] = _digit


def word_to_signature(word):
    """Return the keypad signature of a word: "home" gives "4663".

    A character that is not a letter from a to z becomes a space. The digits
    are collected in a list and joined once at the end; this is Python's
    StringBuffer, because a str can never grow in place.
    """
    digits = []
    for letter in word.lower():
        digits.append(_KEY_OF.get(letter, " "))
    return "".join(digits)


def is_valid_word(word):
    """True when the word is not empty and has only the letters a to z."""
    return len(word) > 0 and all("a" <= c <= "z" for c in word)


def signature_to_words(signature):
    """Return the set of dictionary words (lower case) with this signature.

    Inefficient on purpose: every call opens the file, reads ALL of it and
    computes the signature of every word again.
    """
    result = set()
    with open(DICTIONARY, encoding="utf-8") as f:
        for line in f:
            word = line.strip().lower()
            if is_valid_word(word) and word_to_signature(word) == signature:
                result.add(word)
    return result
```

Listing 12.4 `prototype_check.py`: tests for Problems 12.1 and 12.2 in Python, with the same wrong variants and a timing of three ways to build a string.

```
# prototype_check.py - tests for Problems 12.1 and 12.2, with two tempting
# wrong solutions and a timing of += versus a list joined once.
import sys
import time
import unittest

from predictive.prototype import is_valid_word, signature_to_words, word_to_signature


def word_to_signature_wrong(word):
    """Tempting but wrong: "three letters per key" arithmetic."""
    return "".join(chr(ord("2") + (ord(c) - ord("a")) // 3) for c in word)


def signature_to_words_wrong(signature):
    """Tempting but wrong: a list, no lower-casing, no validity check."""
    with open("words.txt", encoding="utf-8") as f:
        return [line.rstrip("\n") for line in f
                if word_to_signature(line.rstrip("\n")) == signature]


def concat_with_str(n):
    s = ""
    for _ in range(n):
        s = s + "2"
    return s


def concat_with_str_shared(n):
    s = ""
    for _ in range(n):
        keep = s                    # a second reference defeats CPython's shortcut
        s = s + "2"
    return s + keep[:0]


def concat_with_join(n):
    parts = []
    for _ in range(n):
        parts.append("2")
    return "".join(parts)
```

Listing 12.4 (Continued) *prototype_check.py: tests for Problems 12.1 and 12.2 in Python, with the same wrong variants and a timing of three ways to build a string.*

```
class TestPrototype(unittest.TestCase):
    def test_signatures(self):
        self.assertEqual("4663", word_to_signature("home"))
        self.assertEqual("43556", word_to_signature("hello"))
        self.assertEqual("7777 9999", word_to_signature("pqrs wxyz"))
        self.assertEqual("4663", word_to_signature("Home"))
        self.assertEqual("366 8", word_to_signature("don't"))
        self.assertEqual("", word_to_signature(""))

    def test_valid_words(self):
        self.assertTrue(is_valid_word("laundry"))
        self.assertFalse(is_valid_word("e-mail"))
        self.assertFalse(is_valid_word(""))

    def test_lookup(self):
        self.assertEqual(["gone", "good", "goof", "home", "hone", "hood", "hoof"],
                         sorted(signature_to_words("4663")))
        self.assertEqual(["kiss", "lips"], sorted(signature_to_words("5477")))
        self.assertEqual(set(), signature_to_words("22222"))


if __name__ == "__main__":
    print("wrong 'you'      :", word_to_signature_wrong("you"),
          "  right:", word_to_signature("you"))
    print("wrong 4663 list  :", signature_to_words_wrong("4663"))
    for n in (20_000, 40_000, 80_000):
        t0 = time.perf_counter()
        concat_with_str(n)
        t1 = time.perf_counter()
        concat_with_str_shared(n)
        t2 = time.perf_counter()
        concat_with_join(n)
        t3 = time.perf_counter()
        print(f"n = {n:6d}   str +: {(t1 - t0) * 1e3:6.2f} ms   "
              f"str + (shared): {(t2 - t1) * 1e3:7.2f} ms   join: {(t3 - t2) * 1e3:5.2f}
ms")
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

Run it from the `python` folder with `python -m predictive.prototype_check`:

```
wrong 'you'      : :68   right: 968
wrong 4663 list  : ['gone', 'good', 'goof', 'home', 'Home', 'hone', 'hood', 'hoof']
n =  20000   str +:   0.70 ms   str + (shared):    2.60 ms   join:  0.48 ms
n =  40000   str +:   1.23 ms   str + (shared):   15.39 ms   join:  0.96 ms
n =  80000   str +:   2.66 ms   str + (shared):   75.86 ms   join:  2.10 ms
test_lookup (__main__.TestPrototype.test_lookup) ... ok
test_signatures (__main__.TestPrototype.test_signatures) ... ok
test_valid_words (__main__.TestPrototype.test_valid_words) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

The Python timing holds a surprise: the plain `s = s + "2"` loop is *not* quadratic. CPython, the standard interpreter, quietly extends a string in place when nothing else refers to it. The middle column shows how fragile that shortcut is: one extra reference (`keep = s`) switches it off, and the time grows five to six times per doubling of n, the square law again. That is why the Python documentation recommends `join` [4].

Line by line. In **Listing 12.1**, `new StringBuffer(word.length())` reserves exactly enough room. The loop reads each character with `word.charAt(i)`, lowers it with `Character.toLowerCase`, and appends `keyOf` of it. `keyOf` is a switch expression: each `case` lists the letters of one key, the arrow gives the value for them, and `default` yields a space for everything else. `result.toString()` makes one `String` at the end. In **Listing 12.3**, two nested loops at module level fill `_KEY_OF` with 26 entries (the underscore marks it as internal), `_KEY_OF.get(letter, " ")` returns the key or a space, and `"".join(digits)` glues the list together.

A tempting wrong solution. Three letters per key suggests arithmetic: number the letters 0 (`a`) to 25 (`z`) and compute

$$\kappa_{\text{wrong}}(c) = 2 + \left\lfloor \frac{\text{index}(c)}{3} \right\rfloor$$

where $\lfloor x \rfloor$ means "round down". `wordToSignatureWrong` does this, and for `"you"` it prints `:68` instead of `968`. Key 7 carries *four* letters, so from s onward the formula runs one key ahead: s lands on 8, v on 9, and y and z on "10", which as a character is the one after `9`, the colon. It gives the wrong key for exactly four letters, s, v, y and z, passes every test built from "home" and "hello", and is caught only by a test with the four-letter keys, like our check `four-letter keys`. A table cannot make this mistake.

> **CHECK YOURSELF.** What does `wordToSignature` return for `"Laundry"`, and for `"e-mail"`?
>
> *Answer.* `"5286379"` (the capital L is lowered first), and `"3 6245"`: the hyphen becomes a space, so the signature still has six positions.

12.4.2 Problem 12.2 — From a signature to its words, the slow way

> **PROBLEM.** In the same class, write `public static Set<String> signatureToWords(String signature)`. It takes a numeric signature and returns the set of dictionary words that match it, in lower case and without duplicates. The method must use the dictionary file to find the matching words, and it must not store the dictionary in the program. Explain in a comment why this implementation is inefficient. A helper `isValidWord(String word)` that decides whether a line of the dictionary is a proper word is recommended.

In our own words. For every question, open the file, read it from top to bottom, and keep each line that is a proper word with the requested signature. The result is a *set*, so `home` and `Home`, the same word once lowered, appear once.

Design. The return type is a `Set` because `add` on a set that already holds the word does nothing: "no duplicates" is enforced by the type. The file is opened in a try-with-resources statement (Chapter 5), so it is closed whatever happens. And the cost is the whole file: with n words of average length $\bar{m}$, one lookup reads and converts all of them,

$$T_{\text{proto}}(n) = O(n\,\bar{m}) \quad \text{per lookup,} \tag{12.3}$$

and q lookups cost q times as much, because nothing is remembered between calls. That is the inefficiency the comment explains and Parts Two and Three remove.

The solutions are `signatureToWords` in **Listing 12.1** and `signature_to_words` in **Listing 12.3**. Their tests (`4663`, `5477`, `no match`, `test_lookup`) passed above: 4663 gives gone, good, goof, home, hone, hood and hoof, and 5477 gives kiss and lips, a famous nuisance of predictive text.

Line by line. The `Scanner` is created with an explicit `UTF_8` encoding, so `café` reads the same on every system. `hasNextLine` and `nextLine` read line by line; `.trim().toLowerCase()` cleans each line. In `isValidWord(word) && wordToSignature(word).equals(signature)`, the `&&` stops at the first `false`, so invalid lines are never converted, and strings are compared with `equals`, never `==` (Chapter 3). The `catch` turns the checked `IOException` into an unchecked `IllegalStateException`, because a missing dictionary is nothing a caller can repair. `isValidWord` rejects the empty string, then compares every character with the range `'a'` to `'z'`; Python's version is `all("a" <= c <= "z" for c in word)`, and `with open(...)` plays the role of try-with-resources.

A tempting wrong solution. The hurried version, `signatureToWordsWrong`, reads the lines, keeps those whose signature matches, and returns a `List`. Its answer for 4663 contains `Home` next to `home`: the lines were never lowered, and a list keeps both. It also never trims, so `soap` gets a signature with spaces around it and is silently lost. No test with clean, lower-case words finds either fault; a realistic dictionary finds both. A second classic trap lies in closing the file by hand: with `Scanner fileScan = null; try { fileScan = new Scanner(...); ... } finally { fileScan.close(); }`, a missing file leaves `fileScan` as `null`, and the `finally` block throws a `NullPointerException` that hides the real error. Try-with-resources never closes what was never opened.

> **CHECK YOURSELF.** Add one cheap test that lets `signatureToWords` skip most words. Does it change Equation (12.3)?
>
> *Answer.* Test `word.length() == signature.length()` first (a signature has one digit per letter). It skips most conversions, but the whole file is still read, so the cost per lookup stays proportional to n.

12.4.3 Problem 12.3 — Two command-line programs

> **PROBLEM.** Write two command-line programs (classes with a `main` method): `Words2SigProto`, which calls `wordToSignature` for every word given on the command line, and `Sigs2WordsProto`, which calls `signatureToWords` for every signature given on the command line. For example, `Words2SigProto home hello world my name is` should print `input: [home, hello, world, my, name, is]` and `output: 4663 43556 96753 69 6263 47`, and `Sigs2WordsProto` should print one line per signature, the signature followed by its words.

In our own words. The words after the program's name arrive as the array `args` (Java) or the list `sys.argv` (Python). The first program converts each to a signature; the second looks each up and prints one line per signature.

Design. Both are a loop over the arguments. One detail needs care: *reproducible* output. A `HashSet` prints its elements in an order that depends on their hash codes, and Python even changes string hash codes from run to run (*hash randomization*), so we sort before printing: a `TreeSet` in Java, `sorted()` in Python.

The Java solution. Listing 12.5 and **Listing 12.6**.

Listing 12.5 `Words2SigProto.java`: the signature of every word on the command line.

```
// Words2SigProto.java - Problem 12.3: print the signature of every word
// given on the command line.
package predictive;

import java.util.Arrays;

public class Words2SigProto {

    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("USAGE: java predictive.Words2SigProto word1 word2 ...
wordN");
            return;
        }
        System.out.println("input: " + Arrays.toString(args));
        StringBuilder output = new StringBuilder("output:");
        for (String word : args) {
            output.append(' ').append(PredictivePrototype.wordToSignature(word));
        }
        System.out.println(output);
    }
}
```

Listing 12.6 `Sigs2WordsProto.java`: the words of every signature on the command line, one line per signature, in alphabetical order.

```
// Sigs2WordsProto.java - Problem 12.3: print the matching words of every
// signature given on the command line, one line per signature.
package predictive;

import java.util.TreeSet;

public class Sigs2WordsProto {

    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("USAGE: java predictive.Sigs2WordsProto sig1 sig2 ...
sigN");
            return;
        }
        for (String sig : args) {
            // a TreeSet prints the words in alphabetical order, the same on every run
            System.out.println(sig + ": " + new
TreeSet<>(PredictivePrototype.signatureToWords(sig)));
        }
    }
}
```

`java predictive.Words2SigProto home hello world my name is` prints:

```
input: [home, hello, world, my, name, is]
output: 4663 43556 96753 69 6263 47
```

and `java predictive.Sigs2WordsProto 4663 43556 96753 69 6263 47` prints:

```
4663: [gone, good, goof, home, hone, hood, hoof]
43556: [hello]
96753: [world]
69: [my, ow, ox]
6263: [mane, name, oboe]
47: [is]
```

The Python solution. **Listing 12.7** and **Listing 12.8**, run with `python -m predictive.words2sig_proto ...` and `python -m predictive.sigs2words_proto ...`, print exactly the same lines.

Listing 12.7 `words2sig_proto.py`: the same program in Python; the words are `sys.argv[1:]`.

```
# words2sig_proto.py - Problem 12.3: print the signature of every word
# given on the command line.
import sys

from predictive.prototype import word_to_signature


def main(args):
    if not args:
        print("USAGE: python -m predictive.words2sig_proto word1 word2 ... wordN")
        return
    print("input: [" + ", ".join(args) + "]")
    print("output: " + " ".join(word_to_signature(word) for word in args))


if __name__ == "__main__":
    main(sys.argv[1:])                # sys.argv[0] is the program itself
```

Listing 12.8 `sigs2words_proto.py`: the words of every signature on the command line, sorted.

```
# sigs2words_proto.py - Problem 12.3: print the matching words of every
# signature given on the command line, one line per signature.
import sys

from predictive.prototype import signature_to_words


def main(args):
    if not args:
        print("USAGE: python -m predictive.sigs2words_proto sig1 sig2 ... sigN")
        return
    for sig in args:
        # sorted() gives alphabetical order, the same on every run
        print(sig + ": [" + ", ".join(sorted(signature_to_words(sig))) + "]")


if __name__ == "__main__":
    main(sys.argv[1:])
```

```
input: [home, hello, world, my, name, is]
output: 4663 43556 96753 69 6263 47
4663: [gone, good, goof, home, hone, hood, hoof]
43556: [hello]
96753: [world]
69: [my, ow, ox]
6263: [mane, name, oboe]
47: [is]
```

With a quarter-million-word dictionary, such as those shipped with many Linux systems, the same commands also find odd entries, such as `yorke` for 96753; the programs do not change, only the file.

Line by line. The `if` prints a usage message when there are no arguments. `Arrays.toString(args)` formats the array as `[home, hello, ...]`, and the output line grows in a `StringBuilder`. `new TreeSet<>(...)` copies the answer into a sorted set whose `toString()` lists it in order. In Python, `sys.argv[1:]` drops the program's own name, and `", ".join(sorted(...))` builds the bracketed list.

A tempting wrong solution. Printing the arguments directly is wrong in both languages, differently. In Java, `System.out.println("input: " + args)` prints

```
input: [Ljava.lang.String;@659e0bfd
```

because arrays do not override `toString()`: `[L` means "array of", followed by the class name and a hash code that changes from run to run. In Python, `print("input:", sys.argv)` prints

```
input: ['words2sig_wrong.py', 'home', 'hello']
```

with the program's file name first. Python's `sys.argv[0]` is the program; Java's `args[0]` is already the first argument, an off-by-one worth remembering when porting command-line programs.

> **CHECK YOURSELF.** What does `Words2SigProto don't panic` print, and how many arguments does `Sigs2WordsProto` receive if you paste the output line after its name?
>
> *Answer.* `input: [don't, panic]` and `output: 366 8 72642`. The space inside `366 8` separates two arguments, so `Sigs2WordsProto` receives three: punctuation cuts a signature in two, which is why the problem lets us ignore it.

12.5 Part Two: A Sorted List and an Interface

A phone that pauses after every key press feels broken, and the prototype reads the whole dictionary for every question. The remedy: read the dictionary *once*, keep it in memory, and arrange it so that a question can be answered without looking at every word. Part Two uses the simplest such arrangement, a sorted list.

12.5.1 Problem 12.4 — A sorted list of pairs and binary search

> **PROBLEM.** Write a class `DictionaryListImpl`. Its constructor reads the dictionary file and stores it in an `ArrayList`, each entry of which pairs a word with its signature; write a class `WordSig` for these pairs, with `private` fields for the word and the signature, and let it implement `Comparable<WordSig>`. The list must be sorted, once, in the *numerical* order of the signatures (which is not the alphabetical order of the signature strings). The method `signatureToWords` must use binary search, `Collections.binarySearch`, to find the words of a signature. Binary search returns the index of *one* match; the method must return *all* matching words, so scan above and below the index it finds. For example, `signatureToWords("4663")` must give the thirteen or so words that a large dictionary has for 4663, and `signatureToWords("43556")` must give `hello` (and `gekko`, if the dictionary has it).

In our own words. Turn the dictionary into cards, each carrying a signature and a word, and sort them by signature, as a telephone book is sorted by name; cards with the same signature then lie together. To answer a question, find one card with the right signature by repeatedly halving the part of the list where it can be, then collect its neighbors upward and downward while they carry the same signature.

Design. Three ideas make the design.

First, the order. `Collections.sort` and `Collections.binarySearch` ask the elements which comes first, through `Comparable<WordSig>`: `compareTo(WordSig other)` returns a negative number, zero or a positive number [5]. The problem asks for *numerical* order, and that word hides a trap: `String.compareTo` compares character by character, so it puts `"22"` before `"3"`, although $22 > 3$. Converting to a number has a trap of its own (see the wrong solution). The clean cure uses a fact about numbers without leading zeros, and our signatures (digits 2 to 9) never have any: *fewer digits means smaller*, and equal lengths compare exactly as the digit strings do:

$$a < b \Leftrightarrow |a| < |b| \text{ or } (|a| = |b| \text{ and } a <_{\text{str}} b) \tag{12.4}$$

where $|a|$ is the number of digits of a and $<_{\text{str}}$ is the ordinary string order. Equation (12.4) needs no conversion and works for signatures of any length.

Second, the search. Binary search keeps two indices, `low` and `high`, and looks at the middle card: if its signature is smaller than the wanted one, the wanted cards lie to its right (`low` moves past the middle); if larger, to its left (`high` moves before it); if equal, a match is found. Every look halves the part still to be searched [6], so for n cards at most

$$\lceil \log_2(n+1) \rceil \tag{12.5}$$

cards are examined, where $\lceil x \rceil$ means "round up". For our 93 cards that is 7; for 200,000 cards it is 18; for a million, 20. The search never looks at most of the list.

Third, the neighbors. Binary search stops at whichever matching card it hits first. **Table 12.4** shows the real run for 4663 on our dictionary: four looks, ending at index 40, `goof`, in the *middle* of the block of 4663 cards (indices 38 to 45). Walking down finds `good` and `gone`; walking up finds `home` twice (from `home` and `Home`), `hone`, `hood` and `hoof`; the walks stop at `help` (4357) and `iron` (4766).

Table 12.4 Binary search for 4663 among the 93 sorted cards of `words.txt`. Four looks find one match; the others lie at indices 38 to 45, around it.

Look	low	high	Middle	Card in the middle	Decision
1	0	92	46	`4766:iron`	larger than 4663: `high = 45`
2	0	45	22	`228:bat`	smaller (3 digits): `low = 23`
3	23	45	34	`2665:book`	smaller: `low = 35`
4	35	45	40	`4663:goof`	equal: found index 40

Putting it together, building the list costs a sort, and a lookup costs one binary search plus one step per match:

$$T_{\text{build}}(n) = O(n \log n), \qquad T_{\text{list}}(n) = O(\bar{m} \log n + k) \quad \text{per lookup}, \tag{12.6}$$

where k is the number of matching words and $\bar{m}$ the length of a signature, which is what each comparison costs at worst.

The Java solution. **Listing 12.9** is the pair class; **Listing 12.10** the dictionary. `DictionaryListImpl` says `implements Dictionary`; that interface, one method long, is designed in Problem 12.5.

Listing 12.9 `WordSig.java`: a signature and a word, compared in the numerical order of Equation (12.4).

```java
// WordSig.java - Problem 12.4: a word paired with its signature, ordered by
// the NUMERICAL value of the signature.
package predictive;

public class WordSig implements Comparable<WordSig> {

    private final String signature;
    private final String word;

    public WordSig(String signature, String word) {
        this.signature = signature;
        this.word = word;
    }

    public String getSignature() {
        return signature;
    }

    public String getWord() {
        return word;
    }

    /**
     * Numerical order of two digit strings without leading zeros: the shorter
     * number is the smaller one; equal lengths compare digit by digit, which is
     * exactly what String.compareTo does. No conversion to a number is needed,
     * so a signature of any length is compared correctly.
     */
    @Override
    public int compareTo(WordSig other) {
        if (signature.length() != other.signature.length()) {
            return Integer.compare(signature.length(), other.signature.length());
        }
        return signature.compareTo(other.signature);
    }

    @Override
    public String toString() {
        return signature + ":" + word;
    }
}
```

Listing 12.10 `DictionaryListImpl.java`: the dictionary as a sorted `ArrayList<WordSig>`, searched with `Collections.binarySearch` and a walk to both sides.

```
// DictionaryListImpl.java - Problem 12.4: the dictionary in memory as a sorted
// ArrayList of WordSig pairs, searched with binary search.
package predictive;

import java.io.File;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

public class DictionaryListImpl implements Dictionary {

    private final ArrayList<WordSig> dictionary = new ArrayList<>();

    /** Reads every valid word of the file once, then sorts the pairs once. */
    public DictionaryListImpl(String path) throws IOException {
        try (Scanner in = new Scanner(new File(path), StandardCharsets.UTF_8)) {
            while (in.hasNextLine()) {
                String word = in.nextLine().trim().toLowerCase();
                if (PredictivePrototype.isValidWord(word)) {
                    add(word);
                }
            }
        }
        Collections.sort(dictionary);
    }

    private void add(String word) {
        dictionary.add(new WordSig(PredictivePrototype.wordToSignature(word), word));
    }

    @Override
    public Set<String> signatureToWords(String signature) {
        Set<String> result = new HashSet<>();
        // a "dummy" pair: binary search compares signatures only
        int index = Collections.binarySearch(dictionary, new WordSig(signature, ""));
        if (index < 0) {
            return result;                          // not found: -(insertion point) - 1
        }
        // binarySearch finds SOME match; the others are its neighbors
        for (int i = index; i >= 0 && matches(i, signature); i--) {
            result.add(dictionary.get(i).getWord());
        }
```

Listing 12.10 (Continued) *DictionaryListImpl.java: the dictionary as a sorted ArrayList<WordSig>, searched with Collections.binarySearch and a walk to both sides.*

```
        for (int i = index + 1; i < dictionary.size() && matches(i, signature); i++) {
            result.add(dictionary.get(i).getWord());
        }
        return result;
    }

    private boolean matches(int i, String signature) {
        return dictionary.get(i).getSignature().equals(signature);
    }

    /** The number of stored pairs (duplicates in the file are stored twice). */
    public int size() {
        return dictionary.size();
    }
}
```

Listing 12.11 tests the class and demonstrates the two wrong solutions discussed below.

Listing 12.11 `ListCheck.java`: checks for Problem 12.4, the two orders of signatures side by side, and two wrong variants.

```
// ListCheck.java - checks for Problem 12.4, with two tempting wrong solutions:
// keeping only the first binary-search hit, and ordering signatures as doubles.
package predictive;

import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;

public class ListCheck {

    /** The valid words of words.txt as WordSig pairs, sorted by the given order. */
    static List<WordSig> load(Comparator<WordSig> order) throws Exception {
        List<WordSig> list = new ArrayList<>();
        for (String line : Files.readAllLines(Path.of("words.txt"))) {
            String word = line.trim().toLowerCase();
            if (PredictivePrototype.isValidWord(word)) {
                list.add(new WordSig(PredictivePrototype.wordToSignature(word), word));
            }
        }
        list.sort(order);
        return list;
    }

    /** Tempting but wrong: return only the pair that binary search lands on. */
    static Set<String> firstHitOnly(List<WordSig> list, String signature) {
        int i = Collections.binarySearch(list, new WordSig(signature, ""));
        return (i < 0) ? Set.of() : Set.of(list.get(i).getWord());
    }

    /** Tempting but wrong: compare signatures as double values. */
    static final Comparator<WordSig> BY_DOUBLE = (a, b) -> Double.compare(
            Double.parseDouble(a.getSignature()), Double.parseDouble(b.getSignature()));

    private static int passed = 0;
```

Listing 12.11 (Continued) *ListCheck.java: checks for Problem 12.4, the two orders of signatures side by side, and two wrong variants.*

```
    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    private static String sorted(Set<String> words) {
        return new TreeSet<>(words).toString();
    }

    public static void main(String[] args) throws Exception {
        DictionaryListImpl d = new DictionaryListImpl("words.txt");
        check("size", 93, d.size());
        check("4663", "[gone, good, goof, home, hone, hood, hoof]",
sorted(d.signatureToWords("4663")));
        check("43556", "[hello]", sorted(d.signatureToWords("43556")));
        check("5477", "[kiss, lips]", sorted(d.signatureToWords("5477")));
        check("single key", "[a]", sorted(d.signatureToWords("2")));
        check("empty", "[]", sorted(d.signatureToWords("")));
        check("no such key", "[]", sorted(d.signatureToWords("1")));
        check("absent", "[]", sorted(d.signatureToWords("22222")));

        List<WordSig> list = load(Comparator.naturalOrder());
        System.out.println("first pairs      : " + list.subList(0, 8));
        List<WordSig> three = new ArrayList<>(List.of(
                new WordSig("7", "p"), new WordSig("22", "ab"), new WordSig("3", "d")));
        Collections.sort(three);
        System.out.println("numerical order  : " + three);
        three.sort(Comparator.comparing(WordSig::getSignature));
        System.out.println("String order     : " + three);

        System.out.println("first hit only   : " + firstHitOnly(list, "4663"));

        List<WordSig> longs = new ArrayList<>(List.of(
                new WordSig("22222222222222222", "aaaaaaaaaaaaaaaa"),
                new WordSig("22222222222222223", "aaaaaaaaaaaaaaad"),
                new WordSig("22222222222222224", "aaaaaaaaaaaaaaag")));
        longs.sort(BY_DOUBLE);
        System.out.println("as doubles equal : "
                + (BY_DOUBLE.compare(longs.get(0), longs.get(2)) == 0));
        int i = Collections.binarySearch(longs, new WordSig("22222222222222222", ""),
BY_DOUBLE);
        System.out.println("search ...222 by double lands on " + longs.get(i));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
PASS size
PASS 4663
PASS 43556
PASS 5477
PASS single key
PASS empty
PASS no such key
PASS absent
first pairs      : [2:a, 23:be, 43:he, 43:id, 43:if, 46:go, 46:in, 47:is]
numerical order  : [3:d, 7:p, 22:ab]
String order     : [22:ab, 3:d, 7:p]
first hit only   : [goof]
as doubles equal : true
search ...222 by double lands on 22222222222222223:aaaaaaaaaaaaaaad
All 8 checks passed.
```

The line `first pairs` shows the sorted list starting with the one-digit signature of `a`, and the lines `numerical order` and `String order` show the difference Equation (12.4) is about.

The Python solution. Python has no `Comparable`; sorting and the module `bisect` use `<`, which calls the method `__lt__`. `WordSig` in **Listing 12.12** defines it by comparing the tuples `(len(signature), signature)`, which compare element by element: Equation (12.4) in one line. And `bisect.bisect_left` returns the *first* position where the key could be inserted, which for a present key is the first card of its block, so **Listing 12.13** only walks *upward*. **Listing 12.14** repeats the tests and the wrong variants.

Listing 12.12 `word_sig.py`: the pair class in Python; `__lt__` is all that sorting and `bisect` need.

```
# word_sig.py - Problem 12.4: a word paired with its signature, ordered by
# the NUMERICAL value of the signature.


class WordSig:
    __slots__ = ("signature", "word")

    def __init__(self, signature, word):
        self.signature = signature
        self.word = word

    def sort_key(self):
        """Numerical order without conversion: shorter first, then digit by digit."""
        return (len(self.signature), self.signature)

    def __lt__(self, other):              # all that sort() and bisect need
        return self.sort_key() < other.sort_key()

    def __repr__(self):
        return f"{self.signature}:{self.word}"
```

Listing 12.13 `dictionary_list_impl.py`: the sorted list in Python, searched with `bisect_left` and a walk upward.

```python
# dictionary_list_impl.py - Problem 12.4: the dictionary in memory as a sorted
# list of WordSig pairs, searched with binary search.
from bisect import bisect_left

from predictive.dictionary import Dictionary
from predictive.prototype import is_valid_word, word_to_signature
from predictive.word_sig import WordSig


class DictionaryListImpl(Dictionary):

    def __init__(self, path):
        """Read every valid word of the file once, then sort the pairs once."""
        self._dictionary = []
        with open(path, encoding="utf-8") as f:
            for line in f:
                word = line.strip().lower()
                if is_valid_word(word):
                    self._add(word)
        self._dictionary.sort()

    def _add(self, word):
        self._dictionary.append(WordSig(word_to_signature(word), word))

    def signature_to_words(self, signature):
        result = set()
        # bisect_left returns the FIRST position whose pair is not smaller
        i = bisect_left(self._dictionary, WordSig(signature, ""))
        while i < len(self._dictionary) and self._dictionary[i].signature == signature:
            result.add(self._dictionary[i].word)
            i += 1
        return result

    def __len__(self):
        """The number of stored pairs (duplicates in the file are stored twice)."""
        return len(self._dictionary)
```

Listing 12.14 `list_check.py`: tests for Problem 12.4 in Python, with the same two wrong variants.

```
# list_check.py - tests for Problem 12.4, with two tempting wrong solutions:
# keeping only the first binary-search hit, and ordering signatures as floats.
import sys
import unittest
from bisect import bisect_left

from predictive.dictionary_list_impl import DictionaryListImpl
from predictive.prototype import is_valid_word, word_to_signature
from predictive.word_sig import WordSig


def load():
    with open("words.txt", encoding="utf-8") as f:
        words = [line.strip().lower() for line in f]
    return sorted(WordSig(word_to_signature(w), w) for w in words if is_valid_word(w))


def first_hit_only(pairs, signature):
    """Tempting but wrong: return only the pair that binary search lands on."""
    lo, hi = 0, len(pairs) - 1
    while lo <= hi:                       # the textbook search: stops at ANY match
        mid = (lo + hi) // 2
        if pairs[mid].signature == signature:
            return {pairs[mid].word}
        if pairs[mid] < WordSig(signature, ""):
            lo = mid + 1
        else:
            hi = mid - 1
    return set()


def by_float(pair):
    """Tempting but wrong: compare signatures as float values."""
    return float(pair.signature)


class TestListImpl(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.d = DictionaryListImpl("words.txt")

    def test_size(self):
        self.assertEqual(93, len(self.d))
```

Listing 12.14 (Continued) *list_check.py: tests for Problem 12.4 in Python, with the same two wrong variants.*

```
    def test_lookups(self):
        self.assertEqual(["gone", "good", "goof", "home", "hone", "hood", "hoof"],
                         sorted(self.d.signature_to_words("4663")))
        self.assertEqual({"hello"}, self.d.signature_to_words("43556"))
        self.assertEqual({"kiss", "lips"}, self.d.signature_to_words("5477"))
        self.assertEqual({"a"}, self.d.signature_to_words("2"))

    def test_nothing_matches(self):
        for sig in ("", "1", "22222"):
            self.assertEqual(set(), self.d.signature_to_words(sig))


if __name__ == "__main__":
    pairs = load()
    print("first pairs      :", pairs[:8])
    three = sorted([WordSig("7", "p"), WordSig("22", "ab"), WordSig("3", "d")])
    print("numerical order  :", three)
    print("str order        :", sorted(three, key=lambda p: p.signature))
    print("first hit only   :", first_hit_only(pairs, "4663"))
    longs = sorted([WordSig("2222222222222222", "aaaaaaaaaaaaaaaa"),
                    WordSig("2222222222222223", "aaaaaaaaaaaaaaaad"),
                    WordSig("2222222222222224", "aaaaaaaaaaaaaaaag")], key=by_float)
    print("as floats equal  :", by_float(longs[0]) == by_float(longs[2]))
    i = bisect_left(longs, float("2222222222222223"), key=by_float)
    print("search ...223 by float lands on", longs[i])
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
first pairs      : [2:a, 23:be, 43:he, 43:id, 43:if, 46:go, 46:in, 47:is]
numerical order  : [3:d, 7:p, 22:ab]
str order        : [22:ab, 3:d, 7:p]
first hit only   : {'goof'}
as floats equal  : True
search ...223 by float lands on 2222222222222222:aaaaaaaaaaaaaaaa
test_lookups (__main__.TestListImpl.test_lookups) ... ok
test_nothing_matches (__main__.TestListImpl.test_nothing_matches) ... ok
test_size (__main__.TestListImpl.test_size) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Line by line. In **Listing 12.9**, both fields are `private final`: changing a card's signature inside a sorted list would silently break the order binary search relies on. `compareTo` compares lengths with `Integer.compare`, which cannot overflow as subtraction can, and falls back to `String.compareTo` only for equal lengths. In **Listing 12.10**, the constructor reads lines as the prototype did and sorts *once*, after all cards are in; sorting after every card would cost $O(n^2 \log n)$. `new WordSig(signature, "")` is a *probe* whose word does not matter, because `compareTo` looks only at signatures. `Collections.binarySearch` returns a match's index, or $-(\text{insertion point}) - 1$ when there is none (the -1 keeps "not found" negative even for insertion point 0). The first loop walks down from `index`, the second up from `index + 1`, and the `HashSet` reports `home` once. In **Listing 12.13**, `bisect_left` gives the first candidate and the `while` loop collects equal signatures; `__len__` lets us write `len(d)`.

A tempting wrong solution. The first temptation is to stop where binary search stops. `firstHitOnly` does, and answers `[goof]` for 4663: one word of seven. A test with `hello`, alone in its block, would not notice; always test a signature with several words.

The second temptation appears in careful code: compare signatures as numbers, `Double.parseDouble(a.getSignature())` against `Double.parseDouble(b.getSignature())`. A `double` holds about 15 to 16 decimal digits, but signatures are as long as words, and `internationalization` has 20 letters. Beyond 16 digits, different signatures become the *same* `double`: the run shows three 17-digit signatures ending in 2, 3 and 4 that compare `equal`. Binary search then lands on `...223` when asked for `...222`, and our walk, which compares the strings, finds nothing although `...222` is in the list. Python's `float` collapses the same way; `long` merely moves the limit to 18 digits. Equation (12.4) is exact and cheap.

> **CHECK YOURSELF.** In **Table 12.4**, what does the first look decide for the signature 5477 (kiss, lips), and does the answer depend on where the search lands?
>
> *Answer.* Index 46 holds `4766:iron`, smaller than 5477 (same length, `4` before `5`), so `low = 47`, and at most $\lceil \log_2 47 \rceil = 6$ more looks follow. Wherever the search lands in the two-card block, the walk to both sides collects the other card, so the answer is `{kiss, lips}` either way; that is why the walk goes in both directions.

12.5.2 Problem 12.5 — The `Dictionary` interface

> **PROBLEM.** Based on the design of `DictionaryListImpl`, create and document a Java interface `Dictionary`. It will be implemented by the faster dictionaries of Part Three.

In our own words. What does a *user* of a dictionary need? Only to ask "which words match this signature?". How the words are stored is the business of each implementation. The interface should therefore contain that one question and nothing else, and its documentation should say precisely what the answer is in every case, including the awkward ones.

Design. Step 3 of Section 12.2 in action: for every public member of `DictionaryListImpl`, ask whether a *user* needs it. `signatureToWords`: yes. The constructor: no; interfaces have none, and creation is the one place where the concrete class is named. `size()`: no. The `ArrayList<WordSig>`: certainly not, or the map and the tree would have to have one. Then the awkward cases are settled in writing: never `null`, no duplicates, no particular order, and a signature with anything but the digits 2 to 9 matches nothing. Because the tree of Problem 12.8 also returns prefixes, the contract allows that too. A precise contract is what lets three very different classes replace each other.

The Java solution. Listing 12.15.

Listing 12.15 `Dictionary.java`: one method, documented with Javadoc, including the empty and invalid cases.

```java
// Dictionary.java - Problem 12.5: the one promise that every dictionary makes.
package predictive;

import java.util.Set;

/**
 * A dictionary for predictive text: it answers "which words can this row of
 * key presses mean?". Implementations decide how the words are stored.
 */
public interface Dictionary {

    /**
     * Returns the words that match a keypad signature.
     *
     * @param signature the keys pressed, each a digit from 2 to 9, for example "4663"
     * @return the matching lower-case words, without duplicates and in no particular
     *         order; an empty set (never null) when nothing matches or when the
     *         signature contains a character that is not a digit from 2 to 9.
     *         A prefix-matching implementation also returns the first
     *         signature.length() letters of longer words.
     */
    Set<String> signatureToWords(String signature);
}
```

The Python solution. **Listing 12.16** writes the contract as an *abstract base class* (Chapter 7): a class derived from `ABC` whose method is marked `@abstractmethod`. Python refuses to create an object of a class that still has an abstract method, so every implementation must provide `signature_to_words`.

Listing 12.16 `dictionary.py`: the same contract as an abstract base class with a docstring.

```python
# dictionary.py - Problem 12.5: the one promise that every dictionary makes.
from abc import ABC, abstractmethod


class Dictionary(ABC):
    """A dictionary for predictive text: it answers "which words can this row
    of key presses mean?". Subclasses decide how the words are stored."""

    @abstractmethod
    def signature_to_words(self, signature):
        """Return the words that match a keypad signature.

        signature: the keys pressed, each a digit from 2 to 9, e.g. "4663".
        Returns a set of lower-case words (never None): empty when nothing
        matches or when the signature contains a character that is not a
        digit from 2 to 9. A prefix-matching implementation also returns the
        first len(signature) letters of longer words.
        """
```

Line by line. Every method of a Java interface is automatically `public` and `abstract`, so the declaration is the signature and a semicolon; `@param` and `@return` become the familiar Javadoc pages (Chapter 6). An interface with exactly one abstract method is a *functional interface*: a lambda such as `sig -> Set.of("home")` is a complete `Dictionary`, which Problem 12.10 uses to test the GUI's model. In Python, the abstract method's body is only its docstring, which is a legal statement.

A tempting wrong solution. The tempting interface simply copies the public face of the first implementation:

```
public interface Dictionary {
    ArrayList<WordSig> getDictionary();
    int size();
    Set<String> signatureToWords(String signature);
}
```

It compiles, and the list implements it easily. Then the map arrives with no `ArrayList<WordSig>` to return: it must build one on every call or return `null`, breaking every caller that trusted the interface. The first implementation's *representation* has leaked into the contract. Design an interface from the needs of its users, not from the fields of its first implementer [7].

> **CHECK YOURSELF.** Someone proposes to add `String wordToSignature(String word)` to `Dictionary`. Is that a good idea?
>
> *Answer.* No. A word's signature depends on the keypad, not on how a dictionary stores words, so every implementation would contain the same code. It belongs in one shared static helper, `PredictivePrototype.wordToSignature`, as in **Figure 12.2**.

12.5.3 Problem 12.6 — The list program, and the first timing

> **PROBLEM.** Write the command-line program `Sigs2WordsList` for `DictionaryListImpl`, with the same output as `Sigs2WordsProto`. Then compare the time that `Sigs2WordsList` and `Sigs2WordsProto` take to finish with the same large input. Can the difference be made noticeable? Write down the data you used and the timings. On Linux and macOS, the `time` command reports how long a command takes; use its `real` (elapsed) time.

In our own words. Put the same command-line program on top of the list, then race it against the prototype: same questions, same dictionary, many questions, a large dictionary.

Design. The program is **Listing 12.6** with the lookup replaced by a `Dictionary` created on its first line. For the race we need a large dictionary, from `MakeDictionary`, and many signatures, from real text converted by `Words2SigProto`, as a person typing would produce them. To see *where* the time goes, a separate program measures the two phases apart: building the dictionary once, and answering 2,000 lookups.

The Java solution. **Listing 12.17** is the program; its only difference from the Map and Tree versions of Problem 12.9 is the marked line.

Listing 12.17 `Sigs2WordsList.java`: the command-line program on top of any `Dictionary`; this one creates a `DictionaryListImpl`.

```
// Sigs2WordsList.java - Problem 12.6: Sigs2WordsProto with a stored dictionary.
package predictive;

import java.io.IOException;
import java.util.TreeSet;

public class Sigs2WordsList {

    public static void main(String[] args) throws IOException {
        Dictionary d = new DictionaryListImpl(PredictivePrototype.DICTIONARY);   // the one
line that differs
        if (args.length == 0) {
            System.out.println("USAGE: java predictive.Sigs2WordsList sig1 sig2 ... sigN");
            return;
        }
        for (String sig : args) {
            System.out.println(sig + ": " + new TreeSet<>(d.signatureToWords(sig)));
        }
    }
}
```

With `words.txt` it prints the prototype's lines, the first test of the new dictionary:

```
4663: [gone, good, goof, home, hone, hood, hoof]
43556: [hello]
96753: [world]
69: [my, ow, ox]
6263: [mane, name, oboe]
47: [is]
```

Listing 12.18 writes the large dictionary. To produce the *same* random words everywhere and in both languages, it uses its own *linear congruential generator*, which computes each number from the previous one:

$$x_{i+1} = (1103515245\, x_i + 12345) \bmod 2^{31} \tag{12.7}$$

with the seed $x_0 = 12345$ and the constants of the example generator in the C standard. The *low* bits of such generators are notoriously poor: the lowest three repeat with a period of eight [8]. The first version of this program used `next() % 8` to choose word lengths and produced 199,998 words of length 6 out of 200,000; the fix, as in C's `rand()`, is to keep only the high bits, `state >> 16`.

Listing 12.18 `MakeDictionary.java`: 200,000 pronounceable invented words, the same on every computer, written to `big_words.txt`.

```
// MakeDictionary.java - writes a large made-up dictionary for timing experiments.
// The same seed gives the same file in Java and in Python.
package predictive;

import java.io.IOException;
import java.io.PrintWriter;
import java.nio.charset.StandardCharsets;

public class MakeDictionary {

    private static final String CONSONANTS = "bcdfghjklmnpqrstvwxyz";
    private static final String VOWELS = "aeiou";
    private static long state = 12345;               // the seed

    /**
     * A linear congruential generator, as in C's rand(): the next number in
     * 0 .. 32767. Only the high bits are used; the low bits repeat too soon.
     */
    static int next() {
        state = (state * 1103515245L + 12345L) & 0x7fffffffL;
        return (int) (state >> 16);
    }

    /** A pronounceable word of 3 to 10 letters: consonants and vowels alternate. */
    static String nextWord() {
        int length = 3 + next() % 8;
        boolean consonant = next() % 2 == 0;
        StringBuilder sb = new StringBuilder(length);
        for (int i = 0; i < length; i++) {
            sb.append(consonant ? CONSONANTS.charAt(next() % 21) : VOWELS.charAt(next() %
5));
            consonant = !consonant;
        }
        return sb.toString();
    }

    public static void main(String[] args) throws IOException {
        int n = (args.length > 0) ? Integer.parseInt(args[0]) : 200_000;
        String file = (args.length > 1) ? args[1] : "big_words.txt";
        String last = "";
        try (PrintWriter out = new PrintWriter(file, StandardCharsets.UTF_8)) {
            for (int i = 0; i < n; i++) {
                last = nextWord();
                out.println(last);
            }
        }
        System.out.println("wrote " + n + " words to " + file + "; the last one is " +
last);
```

Listing 12.18 (Continued) *MakeDictionary.java: 200,000 pronounceable invented words, the same on every computer, written to big_words.txt.*

```
    }
}
```

```
wrote 200000 words to big_words.txt; the last one is ofad
```

Listing 12.19 measures all four dictionaries of the chapter at once, so Problem 12.9 can reuse it: the time to build each, and the best of three runs of 2,000 lookups. The prototype, which builds nothing, gets only 5 lookups; 2,000 would take minutes.

Listing 12.19 `TimingComparison.java`: build time and lookup time of the four dictionaries on the same file and the same 2,000 signatures.

```java
// TimingComparison.java - Problems 12.6 and 12.9: the four dictionaries measured
// on the same dictionary file and the same signatures.
package predictive;

import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.function.Function;

public class TimingComparison {

    interface Builder {
        Dictionary build(String path) throws Exception;
    }

    /** Best of three runs of all the lookups, in milliseconds; also counts the results. */
    static double timeLookups(Function<String, Set<String>> lookup, List<String> sigs,
long[] found) {
        double best = Double.MAX_VALUE;
        for (int run = 0; run < 3; run++) {
            long count = 0;
            long t0 = System.nanoTime();
            for (String s : sigs) {
                count += lookup.apply(s).size();
            }
            best = Math.min(best, (System.nanoTime() - t0) / 1e6);
            found[0] = count;
        }
        return best;
    }

    static void row(String name, String buildMs, int queries, double lookupMs, long found)
{
        System.out.printf("%-6s %10s %8d %12.2f %12.2f %8d%n",
                name, buildMs, queries, lookupMs, 1000.0 * lookupMs / queries, found);
    }
```

Listing 12.19 (Continued) *TimingComparison.java: build time and lookup time of the four dictionaries on the same file and the same 2,000 signatures.*

```
    public static void main(String[] args) throws Exception {
        String path = PredictivePrototype.DICTIONARY;
        List<String> words = new ArrayList<>();
        for (String line : Files.readAllLines(Path.of(path), StandardCharsets.UTF_8)) {
            String w = line.trim().toLowerCase();
            if (PredictivePrototype.isValidWord(w)) {
                words.add(w);
            }
        }
        List<String> sigs = new ArrayList<>();          // 2000 signatures of real entries
        for (int k = 0; k < 2000; k++) {
            sigs.add(PredictivePrototype.wordToSignature(words.get((int) ((k * 7919L) %
words.size()))));
        }
        System.out.println("dictionary " + path + ": " + words.size() + " words");
        System.out.println("impl     build ms  queries  lookups ms  us/lookup   found");

        long[] found = new long[1];
        List<String> five = sigs.subList(0, 5);          // the prototype is far too slow
for 2000
        double ms = timeLookups(PredictivePrototype::signatureToWords, five, found);
        row("proto", "-", five.size(), ms, found[0]);

        String[] names = {"list", "map", "tree"};
        Builder[] builders = {DictionaryListImpl::new, DictionaryMapImpl::new,
DictionaryTreeImpl::new};
        for (int i = 0; i < names.length; i++) {
            long t0 = System.nanoTime();
            Dictionary d = builders[i].build(path);
            double buildMs = (System.nanoTime() - t0) / 1e6;
            ms = timeLookups(d::signatureToWords, sigs, found);
            row(names[i], String.format("%.1f", buildMs), sigs.size(), ms, found[0]);
        }
    }
}
```

Run with the large dictionary (`PREDICTIVE_DICT=big_words.txt java predictive.TimingComparison`):

```
dictionary big_words.txt: 200000 words
impl     build ms  queries  lookups ms  us/lookup   found
proto           -        5      311.42     62283.18      19
list        419.6     2000        5.41         2.70    5332
map         307.0     2000        0.45         0.22    5332
tree        956.1     2000       27.14        13.57    7496
```

The column `us/lookup` is microseconds per lookup. The prototype needs 62,283, about a sixteenth of a second; the list needs 2.70, more than twenty thousand times less. The list pays once, 420 milliseconds to read and sort 200,000 cards, the price of about seven prototype lookups, and is ahead forever after. The `found` column (5,332 words for list and map) is one more check that they give the same answers.

The Python solution. **Listing 12.20**, **Listing 12.21** and **Listing 12.22** are the same three programs; `make_dictionary.py` writes a file identical to Java's, byte for byte, because both follow Equation (12.7) with the same seed.

Listing 12.20 `sigs2words_list.py`: the list program in Python.

```
# sigs2words_list.py - Problem 12.6: sigs2words_proto with a stored dictionary.
import sys

from predictive.dictionary_list_impl import DictionaryListImpl
from predictive.prototype import DICTIONARY


def main(args):
    d = DictionaryListImpl(DICTIONARY)                  # the one line that differs
    if not args:
        print("USAGE: python -m predictive.sigs2words_list sig1 sig2 ... sigN")
        return
    for sig in args:
        print(sig + ": [" + ", ".join(sorted(d.signature_to_words(sig))) + "]")


if __name__ == "__main__":
    main(sys.argv[1:])
```

```
4663: [gone, good, goof, home, hone, hood, hoof]
43556: [hello]
96753: [world]
69: [my, ow, ox]
6263: [mane, name, oboe]
47: [is]
```

Listing 12.21 `make_dictionary.py`: the same invented dictionary in Python.

```
# make_dictionary.py - writes a large made-up dictionary for timing experiments.
# The same seed gives the same file in Java and in Python.
import sys

CONSONANTS = "bcdfghjklmnpqrstvwxyz"
VOWELS = "aeiou"
_state = 12345                                  # the seed


def next_number():
    """A linear congruential generator, as in C's rand(): the next number in
    0 .. 32767. Only the high bits are used; the low bits repeat too soon."""
    global _state
    _state = (_state * 1103515245 + 12345) & 0x7FFFFFFF
    return _state >> 16


def next_word():
    """A pronounceable word of 3 to 10 letters: consonants and vowels alternate."""
    length = 3 + next_number() % 8
    consonant = next_number() % 2 == 0
    letters = []
    for _ in range(length):
        letters.append(CONSONANTS[next_number() % 21] if consonant else
VOWELS[next_number() % 5])
        consonant = not consonant
    return "".join(letters)


def main(args):
    n = int(args[0]) if len(args) > 0 else 200_000
    file = args[1] if len(args) > 1 else "big_words.txt"
    last = ""
    with open(file, "w", encoding="utf-8", newline="\n") as out:
        for _ in range(n):
            last = next_word()
            out.write(last + "\n")
    print(f"wrote {n} words to {file}; the last one is {last}")


if __name__ == "__main__":
    main(sys.argv[1:])
```

```
wrote 200000 words to big_words.txt; the last one is ofad
```

Listing 12.22 `timing_comparison.py`: the four dictionaries measured in Python.

```
# timing_comparison.py - Problems 12.6 and 12.9: the four dictionaries measured
# on the same dictionary file and the same signatures.
import time

from predictive.dictionary_list_impl import DictionaryListImpl
from predictive.dictionary_map_impl import DictionaryMapImpl
from predictive.dictionary_tree_impl import DictionaryTreeImpl
from predictive.prototype import DICTIONARY, is_valid_word, signature_to_words,
word_to_signature


def time_lookups(lookup, sigs):
    """Best of three runs of all the lookups, in milliseconds; also counts the results."""
    best, found = float("inf"), 0
    for _ in range(3):
        t0 = time.perf_counter()
        found = sum(len(lookup(s)) for s in sigs)
        best = min(best, (time.perf_counter() - t0) * 1e3)
    return best, found


def row(name, build_ms, queries, lookup_ms, found):
    print(f"{name:<6} {build_ms:>10} {queries:8d} {lookup_ms:12.2f} "
          f"{1000.0 * lookup_ms / queries:12.2f} {found:8d}")


def main():
    with open(DICTIONARY, encoding="utf-8") as f:
        words = [w for w in (line.strip().lower() for line in f) if is_valid_word(w)]
    sigs = [word_to_signature(words[(k * 7919) % len(words)]) for k in range(2000)]
    print(f"dictionary {DICTIONARY}: {len(words)} words")
    print("impl     build ms  queries  lookups ms  us/lookup   found")

    five = sigs[:5]                              # the prototype is far too slow for 2000
    ms, found = time_lookups(signature_to_words, five)
    row("proto", "-", len(five), ms, found)

    for name, cls in (("list", DictionaryListImpl), ("map", DictionaryMapImpl),
                      ("tree", DictionaryTreeImpl)):
        t0 = time.perf_counter()
        d = cls(DICTIONARY)
        build_ms = f"{(time.perf_counter() - t0) * 1e3:.1f}"
        ms, found = time_lookups(d.signature_to_words, sigs)
        row(name, build_ms, len(sigs), ms, found)
```

Listing 12.22 (Continued) *timing_comparison.py: the four dictionaries measured in Python.*

```
if __name__ == "__main__":
    main()
```

```
dictionary big_words.txt: 200000 words
impl     build ms  queries  lookups ms  us/lookup   found
proto           -        5     1177.68    235536.63      19
list       1077.7     2000       12.19         6.10    5332
map         373.5     2000        0.72         0.36    5332
tree       1874.9     2000       21.93        10.97    7496
```

The Python numbers are larger, as expected of an interpreter, but the ratio tells the same story: 235,537 against 6.10 microseconds, almost forty thousand.

The race on the command line. The problem asks for the `time` command, which times a whole program, start-up included. We convert the 53 words of `sample_text.txt` (in the chapter's folder) into signatures and give them all to each program:

```
export PREDICTIVE_DICT=big_words.txt
SIGS=$(java predictive.Words2SigProto $(cat sample_text.txt) | sed -n 's/^output: //p')
time java predictive.Sigs2WordsProto $SIGS > /dev/null
time java predictive.Sigs2WordsList  $SIGS > /dev/null
```

The second line keeps only the digits after `output:`, and `> /dev/null` discards the answers, because only the time matters. **Table 12.5** collects the `real` times for all four programs of the chapter. (On Windows, PowerShell's `Measure-Command { ... }` does the same; see the folder's `README.md`.)

Table 12.5 Elapsed time of each command-line program for the 53 signatures of `sample_text.txt` with the 200,000-word dictionary, as reported by `time`. Every run includes starting the program and building its dictionary.

Program	Java, `real`	Python, `real`
`Sigs2WordsProto` / `sigs2words_proto`	4.073 s	15.092 s
`Sigs2WordsList` / `sigs2words_list`	1.022 s	1.031 s
`Sigs2WordsMap` / `sigs2words_map`	0.905 s	0.432 s
`Sigs2WordsTree` / `sigs2words_tree`	2.581 s	1.991 s

So the difference is noticeable: four seconds against one in Java, fifteen against one in Python. But the list's one second is almost all start-up and building; its 53 lookups take about 150 microseconds. `time` cannot separate the phases, which is why **Listing 12.19** measures them apart. The honest summary: *for a handful of questions, building dominates and the prototype is only a few times slower; for thousands of questions, lookups dominate and it is tens of thousands of times slower.*

Line by line. In **Listing 12.18**, `state` is a `long` because $1103515245 \cdot x_i$ overflows an `int`, and `& 0x7fffffffL` keeps the lowest 31 bits, the remainder modulo 2^{31}. `nextWord` alternates consonants and vowels, which makes words pronounceable and their signatures collide as real words do. In **Listing 12.19**, the nested interface `Builder` ("builds a `Dictionary` from a path, possibly throwing") is fitted by the constructor references `DictionaryListImpl::new` and its two siblings, so one loop builds and measures all three. `timeLookups` receives the lookup as a `Function<String, Set<String>>`, here `d::signatureToWords`, and keeps the best of three runs to soften just-in-time compilation. `(k * 7919) % words.size()` spreads the 2,000 picks over the whole dictionary; 7919 is prime, so they do not cycle early.

A tempting wrong solution. The tempting experiment is the statement's example run as printed: `time java predictive.Sigs2WordsList 4663 43556 96753 69 6263 47` with `words.txt`, against the prototype. On the machine used for this book both reported `real 0m0.162s` and `real 0m0.163s`, and one concludes that the list is no faster. What was measured is the start-up of the Java virtual machine; ninety-three words and six questions cannot distinguish the structures. A measurement informs only when the quantity that differs between the candidates dominates everything else.

> **CHECK YOURSELF.** Using the numbers of the Java timing table, after how many lookups does building the list pay off against the prototype, and after how many does it pay off against the map?
>
> *Answer.* Against the prototype: building costs 420 ms and each prototype lookup about 62 ms, so after $420/62 \approx 7$ lookups. Against the map it never pays off: the map built faster (307 ms) *and* answers faster (0.22 µs against 2.70 µs). The list's advantage over the map is elsewhere: it can list signatures in order, which the hash map cannot.

12.6 Part Three: A Map and a Tree

The sorted list answers in a handful of comparisons but has two weaknesses: inserting a word shifts half the cards on average, $O(n)$ work, and it answers only *whole* signatures, while a predictive phone should suggest words as they are typed. Part Three removes both.

12.6.1 Problem 12.7 — A multi-valued map

> **PROBLEM.** Write a class `DictionaryMapImpl` that implements `Dictionary` and stores the dictionary in a generic *multi-valued map*: a map from each signature to the collection of its words. Like the list, it must find the words of a signature quickly, but without scanning to either side of an index; unlike the list, it must also allow new words to be inserted efficiently. Choose a `Map` implementation from the Java Collections API, explain how it works, and justify the choice. The constructor fills the map; `signatureToWords` returns, as a `Set<String>`, only the whole words whose length equals the signature's. Do not use `WordSig`.

In our own words. Instead of a row of cards, keep a cabinet of drawers, each labeled with a signature and holding its words. To answer, open the drawer with the right label; to add a word, find its drawer (making one if needed) and drop the word in.

Design. The type says it all: `Map<String, Set<String>>`, from a signature to a set of words. The value is a *set* because one signature has several words and a word must not appear twice in a drawer. Which `Map`? Java offers two main implementations [5]:

- `HashMap` computes a number from the key, its *hash code*, which picks a *bucket* in an internal array; a lookup goes to that bucket and compares the few keys found there. On average this costs the same whatever the size of the map, $O(1)$ *expected* [6], if the hash codes are well spread, as those of `String` are. Keys come out in no useful order.
- `TreeMap` keeps its keys in a balanced search tree ordered by `compareTo`: $O(\log n)$ per lookup or insertion, and the keys come out sorted, which also answers questions such as "all keys between these two".

We only ever ask "what is in the drawer labeled s?" and "put this word in its drawer", and neither needs order, so `HashMap` it is. (Exercise 12.3 shows what `TreeMap` would buy instead.) Counting the hashing of the signature's $\bar{m}$ characters,

$$T_{\text{map}}(n) = O(\bar{m} + k) \text{ expected per lookup}, \qquad T_{\text{insert}} = O(\bar{m}) \text{ expected}, \tag{12.8}$$

and neither depends on n. A missing drawer gives an empty set, as the contract of **Listing 12.15** demands, and the drawer itself is never handed out, only a read-only view of it; the wrong solution shows why.

The Java solution. **Listing 12.23** is the dictionary; **Listing 12.24** tests it.

Listing 12.23 `DictionaryMapImpl.java`: a `HashMap` from signatures to sets of words, with a public `addWord` for fast insertion and a read-only result.

```
// DictionaryMapImpl.java - Problem 12.7: the dictionary as a multi-valued map,
// signature -> set of words, kept in a HashMap.
package predictive;

import java.io.File;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Scanner;
import java.util.Set;

public final class DictionaryMapImpl implements Dictionary {

    // HashMap: a lookup or an insertion costs O(1) expected time, whatever the
    // number of signatures; we never need the signatures in sorted order.
    private final Map<String, Set<String>> dictionary = new HashMap<>();

    public DictionaryMapImpl(String path) throws IOException {
        try (Scanner in = new Scanner(new File(path), StandardCharsets.UTF_8)) {
            while (in.hasNextLine()) {
                String word = in.nextLine().trim().toLowerCase();
                if (PredictivePrototype.isValidWord(word)) {
                    addWord(word);
                }
            }
        }
    }

    /** Inserts one word: find (or create) the set of its signature, then add. */
    public void addWord(String word) {
        String signature = PredictivePrototype.wordToSignature(word);
        dictionary.computeIfAbsent(signature, s -> new HashSet<>()).add(word);
    }

    @Override
    public Set<String> signatureToWords(String signature) {
        Set<String> words = dictionary.get(signature);
        if (words == null) {
            return Collections.emptySet();
        }
        return Collections.unmodifiableSet(words);   // a read-only view, no copying
    }
```

Listing 12.23 (Continued) *DictionaryMapImpl.java: a HashMap from signatures to sets of words, with a public addWord for fast insertion and a read-only result.*

```
    /** The number of different signatures (keys of the map). */
    public int signatureCount() {
        return dictionary.size();
    }
}
```

Listing 12.24 `MapCheck.java`: checks for Problem 12.7, including an attempt to clear a returned set, and a map class with two wrong methods.

```
// MapCheck.java - checks for Problem 12.7, with two tempting wrong solutions:
// put() that overwrites, and a getter that hands out the map's own set.
package predictive;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;

public class MapCheck {

    /** Tempting but wrong, twice over. */
    static final class LeakyMap {
        private final Map<String, Set<String>> map = new HashMap<>();

        void addWordOverwriting(String word) {        // wrong 1: a new set every time
            Set<String> s = new HashSet<>();
            s.add(word);
            map.put(PredictivePrototype.wordToSignature(word), s);
        }

        void addWord(String word) {
            map.computeIfAbsent(PredictivePrototype.wordToSignature(word), k -> new
HashSet<>()).add(word);
        }

        Set<String> signatureToWords(String sig) {    // wrong 2: the map's own set
            return map.getOrDefault(sig, new HashSet<>());
        }
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }
```

Listing 12.24 (Continued) *MapCheck.java: checks for Problem 12.7, including an attempt to clear a returned set, and a map class with two wrong methods.*

```
    public static void main(String[] args) throws Exception {
        DictionaryMapImpl d = new DictionaryMapImpl("words.txt");
        check("signatures", 61, d.signatureCount());
        check("4663", "[gone, good, goof, home, hone, hood, hoof]",
                new TreeSet<>(d.signatureToWords("4663")).toString());
        check("absent", "[]", d.signatureToWords("22222").toString());
        check("whole words only", "[work]", d.signatureToWords("9675").toString());
        d.addWord("hoop");
        check("added hoop", true, d.signatureToWords("4667").contains("hoop"));
        try {
            d.signatureToWords("4663").clear();
            System.out.println("clear() was allowed");
        } catch (UnsupportedOperationException e) {
            System.out.println("clear() refused: UnsupportedOperationException");
        }
        check("still 7 words", 7, d.signatureToWords("4663").size());

        LeakyMap leaky = new LeakyMap();
        for (String w : new String[] {"gone", "good", "home"}) {
            leaky.addWordOverwriting(w);
        }
        System.out.println("overwriting put  : " + leaky.signatureToWords("4663"));
        LeakyMap leaky2 = new LeakyMap();
        for (String w : new String[] {"gone", "good", "home"}) {
            leaky2.addWord(w);
        }
        leaky2.signatureToWords("4663").clear();       // a caller "just tidying up"
        System.out.println("after caller clear: " + leaky2.signatureToWords("4663"));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
PASS signatures
PASS 4663
PASS absent
PASS whole words only
PASS added hoop
clear() refused: UnsupportedOperationException
PASS still 7 words
overwriting put  : [home]
after caller clear: []
All 6 checks passed.
```

The 93 valid lines produce 61 drawers. The check `whole words only` shows that the map, like the list, answers only complete signatures (`work` for 9675, nothing of `world`), and `clear() refused` shows the read-only view at work.

The Python solution. Python's `dict` is a hash map. `setdefault(key, default)` returns the value under `key`, storing `default` first if the key is absent: "find or create the drawer". The result is returned as a `frozenset`, an immutable copy; **Listing 12.26** holds the tests.

Listing 12.25 `dictionary_map_impl.py`: the multi-valued map in Python, returning a `frozenset`.

```python
# dictionary_map_impl.py - Problem 12.7: the dictionary as a multi-valued map,
# signature -> set of words, kept in a dict (Python's hash map).
from predictive.dictionary import Dictionary
from predictive.prototype import is_valid_word, word_to_signature


class DictionaryMapImpl(Dictionary):

    def __init__(self, path):
        # dict: a lookup or an insertion costs O(1) expected time
        self._dictionary = {}
        with open(path, encoding="utf-8") as f:
            for line in f:
                word = line.strip().lower()
                if is_valid_word(word):
                    self.add_word(word)

    def add_word(self, word):
        """Insert one word: find (or create) the set of its signature, then add."""
        self._dictionary.setdefault(word_to_signature(word), set()).add(word)

    def signature_to_words(self, signature):
        # a frozenset is a read-only copy: callers cannot change the dictionary
        return frozenset(self._dictionary.get(signature, ()))

    def signature_count(self):
        """The number of different signatures (keys of the dict)."""
        return len(self._dictionary)
```

Listing 12.26 `map_check.py`: tests for Problem 12.7 in Python, and the same two wrong methods.

```
# map_check.py - tests for Problem 12.7, with two tempting wrong solutions:
# an assignment that overwrites, and a getter that hands out the dict's own set.
import sys
import unittest

from predictive.dictionary_map_impl import DictionaryMapImpl
from predictive.prototype import word_to_signature


class LeakyMap:
    """Tempting but wrong, twice over."""

    def __init__(self):
        self.map = {}

    def add_word_overwriting(self, word):           # wrong 1: a new set every time
        self.map[word_to_signature(word)] = {word}

    def add_word(self, word):
        self.map.setdefault(word_to_signature(word), set()).add(word)

    def signature_to_words(self, sig):              # wrong 2: the dict's own set
        return self.map.get(sig, set())


class TestMapImpl(unittest.TestCase):
    def setUp(self):
        self.d = DictionaryMapImpl("words.txt")

    def test_counts_and_lookups(self):
        self.assertEqual(61, self.d.signature_count())
        self.assertEqual(["gone", "good", "goof", "home", "hone", "hood", "hoof"],
                         sorted(self.d.signature_to_words("4663")))
        self.assertEqual(frozenset(), self.d.signature_to_words("22222"))
        self.assertEqual({"work"}, self.d.signature_to_words("9675"))

    def test_add_word(self):
        self.d.add_word("hoop")
        self.assertIn("hoop", self.d.signature_to_words("4667"))

    def test_result_is_read_only(self):
        with self.assertRaises(AttributeError):     # frozenset has no clear()
            self.d.signature_to_words("4663").clear()
        self.assertEqual(7, len(self.d.signature_to_words("4663")))
```

Listing 12.26 (Continued) *map_check.py: tests for Problem 12.7 in Python, and the same two wrong methods.*

```
if __name__ == "__main__":
    leaky = LeakyMap()
    for w in ("gone", "good", "home"):
        leaky.add_word_overwriting(w)
    print("overwriting =     :", leaky.signature_to_words("4663"))
    leaky2 = LeakyMap()
    for w in ("gone", "good", "home"):
        leaky2.add_word(w)
    leaky2.signature_to_words("4663").clear()       # a caller "just tidying up"
    print("after caller clear:", leaky2.signature_to_words("4663"))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
overwriting =     : {'home'}
after caller clear: set()
test_add_word (__main__.TestMapImpl.test_add_word) ... ok
test_counts_and_lookups (__main__.TestMapImpl.test_counts_and_lookups) ... ok
test_result_is_read_only (__main__.TestMapImpl.test_result_is_read_only) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Line by line. In **Listing 12.23**, the class is `final` because its constructor calls the public method `addWord`: in a subclass, an overriding `addWord` could run before the subclass's fields exist [7]. `computeIfAbsent(signature, s -> new HashSet<>())` returns the drawer, creating an empty `HashSet` first if there is none, and `.add(word)` drops the word in. `get` returns `null` for a missing key, which we turn into the shared immutable `Collections.emptySet()`. `Collections.unmodifiableSet(words)` wraps the drawer in a *view* that reads through to the real set and throws `UnsupportedOperationException` from every changing method; making it costs $O(1)$. Python's library has no such view for sets, so **Listing 12.25** copies the k words into a `frozenset`, still cheap.

A tempting wrong solution. `LeakyMap` in both check programs contains the two mistakes that this problem invites. The first is in insertion. The method `addWordOverwriting` does, in effect,

```
map.put(signature, new HashSet<>(Set.of(word)));
```

for every word, which *replaces* the drawer, so only the last word of each signature survives: after gone, good and home, the drawer 4663 holds `[home]`. The second is in the lookup: returning the map's own set hands every caller a key to the cabinet. A caller that "just tidies up" by calling `clear()` on its result empties the drawer *inside the dictionary*, and the next lookup of 4663 finds nothing, although no dictionary method was called in between. As in Chapter 8's encapsulation problem: never hand out a mutable internal object; hand out a view or a copy.

> **CHECK YOURSELF.** Our `signatureToWords` returns an *unmodifiable view* in Java and a *copy* in Python. Name one thing a caller can observe that differs between the two.
>
> *Answer.* A view reads through to the real drawer, so if someone calls `addWord` after the caller received the view, the view shows the new word too. A copy is a snapshot and does not change. (Both refuse modification by the caller: the view throws `UnsupportedOperationException`, and a `frozenset` has no `clear` or `add` at all.)

12.6.2 Problem 12.8 — A tree of keys that also matches prefixes

> **PROBLEM.** Write a class `DictionaryTreeImpl` that implements `Dictionary` and stores the dictionary in a tree of your own. Searching must be fast (like the list) and inserting must be fast (like the map). In addition, the tree must find words when only the first part of the signature, a *prefix*, is known, so that users can see the word they intend while they are still typing it. Each node has up to eight branches, one for each key 2 to 9; the path from the root to a node spells a signature or the beginning of one; each node stores the collection of words whose signature begins with that path. The root stores no words. The constructor takes the path of the dictionary file. `signatureToWords` returns, as a `Set<String>`, the matching words and prefixes of words, each cut to the length of the signature. To save memory, store whole words as read from the file, and write a helper that trims all the words of a collection.

In our own words. Picture roads. From the root, eight roads leave, labeled 2 to 9; each leads to a crossing where up to eight roads leave again. The word `good` is stored by driving 4, 6, 6, 3 and

leaving a copy at every crossing passed: `4`, `46`, `466` and `4663`. To answer "4663", drive the same roads and look at what was left at the last crossing. Because words are left at *every* crossing, the crossing `466` knows `good`, `gone` and `home` although their signatures are longer: that is how prefixes are matched. Cut to the length of the question, `good` and `goof` both become `goo`, and the set keeps one.

Design. **Figure 12.3** draws the tree for eight words: act, bat, be, cat, go, gone, good and in. The labels on the edges are keys; each box shows the path that leads to it and the words stored there.

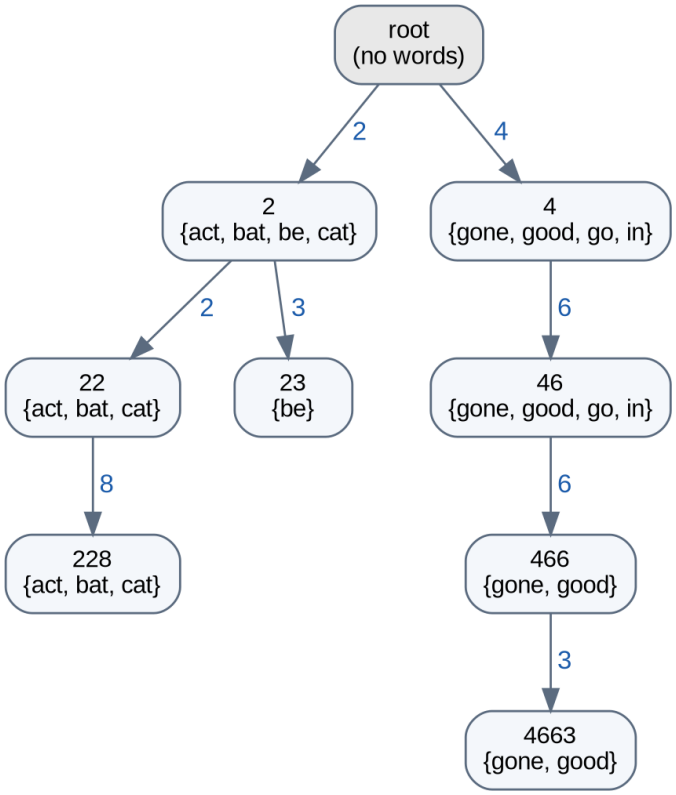


Figure 12.3 The tree dictionary for eight words. Each word is stored in every node on the path of its signature, so the node `46` holds all four words that begin with the keys 4 and 6.

Read three questions off the figure: `"228"` finds `{act, bat, cat}`; `"46"` finds `{gone, good, go, in}`, which cut to two letters become `{go, in}`; and `"229"` finds no road 9 after 22, so the answer is empty.

Such a tree, whose paths spell keys, is a *trie* (from re*trie*val) [9]. It is recursive like the binary tree of Chapter 8, a node being a set of words plus eight children that are `null` or nodes, and its methods follow the *delegation recipe* of Section 8.2, as **Table 12.6** shows.

Table 12.6 The delegation recipe of Chapter 8 applied to the two recursive methods of the tree dictionary.

Question	`insert(signature, pos, word)`	`find(signature, pos)`
Base case	`pos == length`: nothing left to do	`pos == length`: the answer is this node
My own job	create the child for key `signature[pos]` if missing; add `word` to it	look up the child for key `signature[pos]`; `null` if missing or not a key
Delegation	`child.insert(signature, pos + 1, word)`	`child.find(signature, pos + 1)`

Key `'2'` goes to child 0 and `'9'` to child 7. Inserting a word of m letters visits m nodes, and finding the node of a signature of length ℓ visits ℓ, whatever the size of the dictionary. But then the

answer is built: every word stored at that node is cut and added to the result, and nodes near the root store many words. With W_s words at the node of s,

$$T_{\text{tree}}(s) = O(\ell + W_s\,\ell) \text{ per lookup}, \qquad \text{memory} = O\left(\sum_{w} |w|\right) \text{ word references}, \tag{12.9}$$

because each word w is stored in the $|w|$ nodes on its path. Our 90 distinct words have 354 letters, so 354 references in 120 nodes; the 165,929 distinct invented words need 1,183,435. The strings are not copied: every node refers to the same `String` object, which is the memory saving the problem asks for.

The Java solution. Listing 12.27.

Listing 12.27 `DictionaryTreeImpl.java`: a trie with eight children per node, filled and searched by delegation; results are trimmed to the length of the signature.

```
// DictionaryTreeImpl.java - Problem 12.8: the dictionary as a tree with up to
// eight branches per node, one for each key 2..9; it also matches prefixes.
package predictive;

import java.io.File;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.Collections;
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

public final class DictionaryTreeImpl implements Dictionary {

    // the whole words whose signature starts with the path from the root to here
    private final Set<String> words = new HashSet<>();
    // children[0] is the branch for key '2', ..., children[7] for key '9'
    private final DictionaryTreeImpl[] children = new DictionaryTreeImpl[8];

    /** Builds the root and inserts every valid word of the file. */
    public DictionaryTreeImpl(String path) throws IOException {
        try (Scanner in = new Scanner(new File(path), StandardCharsets.UTF_8)) {
            while (in.hasNextLine()) {
                String word = in.nextLine().trim().toLowerCase();
                if (PredictivePrototype.isValidWord(word)) {
                    insert(word);
                }
            }
        }
    }

    private DictionaryTreeImpl() {                // an empty inner node
    }

    /** Inserts one word along the path of its signature. The root stores nothing. */
    public void insert(String word) {
        insert(PredictivePrototype.wordToSignature(word), 0, word);
    }
```

Listing 12.27 (Continued) *DictionaryTreeImpl.java: a trie with eight children per node, filled and searched by delegation; results are trimmed to the length of the signature.*

```
    private void insert(String signature, int pos, String word) {
        if (pos == signature.length()) {
            return;                                 // the whole signature is used up
        }
        int i = childIndex(signature.charAt(pos));
        if (children[i] == null) {
            children[i] = new DictionaryTreeImpl();
        }
        children[i].words.add(word);                // my child's path is a prefix of word
        children[i].insert(signature, pos + 1, word);    // delegation: the rest is its job
    }

    /** The node at the end of the path spelled by the signature, or null. */
    private DictionaryTreeImpl find(String signature, int pos) {
        if (pos == signature.length()) {
            return this;
        }
        int i = childIndex(signature.charAt(pos));
        if (i < 0 || children[i] == null) {
            return null;                            // no such key, or no word goes this way
        }
        return children[i].find(signature, pos + 1);
    }

    private static int childIndex(char key) {
        return (key >= '2' && key <= '9') ? key - '2' : -1;
    }

    @Override
    public Set<String> signatureToWords(String signature) {
        DictionaryTreeImpl node = find(signature, 0);
        if (node == null) {
            return Collections.emptySet();
        }
        return trim(node.words, signature.length());
    }

    /** The first n letters of every word; equal prefixes collapse into one. */
    private static Set<String> trim(Set<String> words, int n) {
        Set<String> result = new HashSet<>();
        for (String w : words) {
            result.add(w.substring(0, n));
        }
        return result;
    }
```

Listing 12.27 (Continued) *DictionaryTreeImpl.java: a trie with eight children per node, filled and searched by delegation; results are trimmed to the length of the signature.*

```
    /** The number of nodes, the root included. */
    public int nodeCount() {
        int count = 1;
        for (DictionaryTreeImpl child : children) {
            if (child != null) {
                count += child.nodeCount();
            }
        }
        return count;
    }
}
```

Listing 12.28 tests whole words and prefixes, and demonstrates the wrong solution below.

Listing 12.28 `TreeCheck.java`: checks for Problem 12.8 and a tree that stores words only at the ends of their paths.

```
// TreeCheck.java - checks for Problem 12.8, with the tempting wrong solution that
// stores each word only at the end of its path.
package predictive;

import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;

public class TreeCheck {

    /** Tempting but wrong: a word is stored only in the LAST node of its path. */
    static final class EndOnlyTree {
        private final Set<String> words = new HashSet<>();
        private final EndOnlyTree[] children = new EndOnlyTree[8];

        void insert(String word) {
            EndOnlyTree node = this;
            for (char key : PredictivePrototype.wordToSignature(word).toCharArray()) {
                int i = key - '2';
                if (node.children[i] == null) {
                    node.children[i] = new EndOnlyTree();
                }
                node = node.children[i];
            }
            node.words.add(word);
        }

        Set<String> signatureToWords(String signature) {
            EndOnlyTree node = this;
            for (char key : signature.toCharArray()) {
                node = node.children[key - '2'];
                if (node == null) {
                    return Set.of();
                }
            }
            return node.words;
        }
    }

    private static int passed = 0;
```

Listing 12.28 (Continued) *TreeCheck.java: checks for Problem 12.8 and a tree that stores words only at the ends of their paths.*

```
    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    private static String sorted(Set<String> words) {
        return new TreeSet<>(words).toString();
    }

    public static void main(String[] args) throws Exception {
        DictionaryTreeImpl d = new DictionaryTreeImpl("words.txt");
        check("4663", "[gone, good, goof, home, hone, hood, hoof]",
sorted(d.signatureToWords("4663")));
        check("96753", "[worke, world]", sorted(d.signatureToWords("96753")));
        check("9675", "[work, worl]", sorted(d.signatureToWords("9675")));
        check("46", "[go, ho, in]", sorted(d.signatureToWords("46")));
        check("4", "[g, h, i]", sorted(d.signatureToWords("4")));
        check("empty", "[]", sorted(d.signatureToWords("")));
        check("no such key", "[]", sorted(d.signatureToWords("41")));
        check("too long", "[]", sorted(d.signatureToWords("4663333")));
        System.out.println("nodes            : " + d.nodeCount());
        d.insert("hoop");
        check("insert hoop", "[hoop]", sorted(d.signatureToWords("4667")));

        EndOnlyTree wrong = new EndOnlyTree();
        for (String w : new String[] {"work", "worked", "world", "worlds"}) {
            wrong.insert(w);
        }
        System.out.println("end-only 9675    : " + sorted(wrong.signatureToWords("9675")));
        System.out.println("end-only 96753   : " +
sorted(wrong.signatureToWords("96753")));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
PASS 4663
PASS 96753
PASS 9675
PASS 46
PASS 4
PASS empty
PASS no such key
PASS too long
nodes            : 121
PASS insert hoop
end-only 9675    : [work]
end-only 96753   : [world]
All 9 checks passed.
```

The answers agree with the list and the map for 4663; for 96753 the tree adds `worke` (from `worked` and `worker`), for 9675 it gives `work` and `worl`, and for the single key 4 the letters `g`, `h` and `i`. A signature longer than every path finds no node (`too long`).

The Python solution. **Listing 12.29** separates the node, a small private class `_Node`, from the public `DictionaryTreeImpl`, which holds the root. The recursion lives in `_Node`, and **Listing 12.30** holds the tests.

Listing 12.29 `dictionary_tree_impl.py`: the same trie in Python; `_Node` does the recursion and `_trim` is a set comprehension.

```
# dictionary_tree_impl.py - Problem 12.8: the dictionary as a tree with up to
# eight branches per node, one for each key 2..9; it also matches prefixes.
from predictive.dictionary import Dictionary
from predictive.prototype import is_valid_word, word_to_signature


def _child_index(key):
    return ord(key) - ord("2") if "2" <= key <= "9" else -1


class _Node:
    __slots__ = ("words", "children")

    def __init__(self):
        self.words = set()           # whole words whose signature starts with my path
        self.children = [None] * 8   # children[0] is key '2', ..., children[7] is '9'

    def insert(self, signature, pos, word):
        if pos == len(signature):
            return                                  # the whole signature is used up
        i = _child_index(signature[pos])
        if self.children[i] is None:
            self.children[i] = _Node()
        self.children[i].words.add(word)            # my child's path is a prefix of word
        self.children[i].insert(signature, pos + 1, word)   # delegation

    def find(self, signature, pos):
        """The node at the end of the path spelled by the signature, or None."""
        if pos == len(signature):
            return self
        i = _child_index(signature[pos])
        if i < 0 or self.children[i] is None:
            return None
        return self.children[i].find(signature, pos + 1)

    def node_count(self):
        return 1 + sum(c.node_count() for c in self.children if c is not None)


class DictionaryTreeImpl(Dictionary):
```

Listing 12.29 (Continued) *dictionary_tree_impl.py: the same trie in Python; _Node does the recursion and _trim is a set comprehension.*

```
    def __init__(self, path):
        """Build the root and insert every valid word of the file."""
        self._root = _Node()                          # the root stores no words
        with open(path, encoding="utf-8") as f:
            for line in f:
                word = line.strip().lower()
                if is_valid_word(word):
                    self.insert(word)

    def insert(self, word):
        self._root.insert(word_to_signature(word), 0, word)

    def signature_to_words(self, signature):
        node = self._root.find(signature, 0)
        if node is None:
            return set()
        return self._trim(node.words, len(signature))

    @staticmethod
    def _trim(words, n):
        """The first n letters of every word; equal prefixes collapse into one."""
        return {w[:n] for w in words}

    def node_count(self):
        return self._root.node_count()
```

Listing 12.30 `tree_check.py`: tests for Problem 12.8 in Python, and the end-only tree.

```
# tree_check.py - tests for Problem 12.8, with the tempting wrong solution that
# stores each word only at the end of its path.
import sys
import unittest

from predictive.dictionary_tree_impl import DictionaryTreeImpl
from predictive.prototype import word_to_signature


class EndOnlyTree:
    """Tempting but wrong: a word is stored only in the LAST node of its path."""

    def __init__(self):
        self.words = set()
        self.children = [None] * 8

    def insert(self, word):
        node = self
        for key in word_to_signature(word):
            i = ord(key) - ord("2")
            if node.children[i] is None:
                node.children[i] = EndOnlyTree()
            node = node.children[i]
        node.words.add(word)

    def signature_to_words(self, signature):
        node = self
        for key in signature:
            node = node.children[ord(key) - ord("2")]
            if node is None:
                return set()
        return node.words


class TestTreeImpl(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.d = DictionaryTreeImpl("words.txt")

    def check(self, expected, signature):
        self.assertEqual(expected, sorted(self.d.signature_to_words(signature)))
```

Listing 12.30 (Continued) *tree_check.py: tests for Problem 12.8 in Python, and the end-only tree.*

```
    def test_whole_words_and_prefixes(self):
        self.check(["gone", "good", "goof", "home", "hone", "hood", "hoof"], "4663")
        self.check(["worke", "world"], "96753")
        self.check(["work", "worl"], "9675")
        self.check(["go", "ho", "in"], "46")
        self.check(["g", "h", "i"], "4")

    def test_nothing_matches(self):
        for sig in ("", "41", "4663333"):
            self.check([], sig)

    def test_insert(self):
        d = DictionaryTreeImpl("words.txt")
        d.insert("hoop")
        self.assertEqual({"hoop"}, d.signature_to_words("4667"))


if __name__ == "__main__":
    print("nodes            :", DictionaryTreeImpl("words.txt").node_count())
    wrong = EndOnlyTree()
    for w in ("work", "worked", "world", "worlds"):
        wrong.insert(w)
    print("end-only 9675    :", sorted(wrong.signature_to_words("9675")))
    print("end-only 96753   :", sorted(wrong.signature_to_words("96753")))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
nodes            : 121
end-only 9675    : ['work']
end-only 96753   : ['world']
test_insert (__main__.TestTreeImpl.test_insert) ... ok
test_nothing_matches (__main__.TestTreeImpl.test_nothing_matches) ... ok
test_whole_words_and_prefixes (__main__.TestTreeImpl.test_whole_words_and_prefixes) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Both languages build the same 121 nodes (the root and 120 others) for our dictionary.

Line by line. In **Listing 12.27**, the two fields are the whole node: `words`, a `HashSet<String>`, and `children`, eight references that start as `null`. The private constructor makes an empty inner node that no outsider can create. The public `insert(String word)` computes the signature once and starts the private recursive `insert` at position 0 of the root; that method is **Table 12.6**, column one, and it adds the word to the *child*, never to `this`, so the root stays empty. `find` is column two. `trim` builds a new set of `w.substring(0, n)`; words sharing a prefix collapse because the result is a set. `nodeCount` is one more delegation. In Python, `[None] * 8` makes the eight empty children, `ord(key) - ord("2")` is the child index, and `{w[:n] for w in words}`, a *set comprehension*, is the whole `trim`.

A tempting wrong solution. The tempting design stores each word only where its path *ends*, as a map would. `EndOnlyTree` answers whole signatures (`world` for 96753), but for the prefix 9675, halfway through typing `world`, it answers only `[work]`: `world`, `worked` and `worlds` pass through without a trace, and the point of the tree is lost. The opposite mistake, adding the word to `this` as well, fills the root with the whole dictionary and makes `signatureToWords("")` return `{""}`; hence the rule “the root stores no words”.

> **CHECK YOURSELF.** Insert only `home` and `hood` into an empty tree. How many nodes does it have (root included), and what does `signatureToWords("466")` return?
>
> *Answer.* The paths are 4-6-6-3 and 4-6-6-3 again (`hood` has the signature 4663 too), so there are 4 nodes below the root, 5 in total, and each of the four stores `{home, hood}`. Cut to three letters they become `hom` and `hoo`, so the answer is `{hom, hoo}`.

12.6.3 Problem 12.9 — One line to change, and the full race

> **PROBLEM.** It should be possible to change just one line of `Sigs2WordsList` so that it works with any implementation of `Dictionary`. Write `Sigs2WordsMap` and `Sigs2WordsTree` that way. Compare the time that `Sigs2WordsMap` and `Sigs2WordsTree` take with large inputs. Can the difference between `Sigs2WordsList` and `Sigs2WordsMap`, or between `Sigs2WordsTree` and `Sigs2WordsMap`, be made noticeable? Write down the data and the timings.

In our own words. The payoff of Problem 12.5: `Sigs2WordsList` declares `Dictionary d` and uses only `d.signatureToWords`, so the class created on its first line is its only tie to the list. Then the race of Problem 12.6 is run with all four contestants.

The Java solution. **Listing 12.31** and **Listing 12.32** differ from **Listing 12.17** in the marked line (and, of course, in the class name and the usage message).

Listing 12.31 `Sigs2WordsMap.java`: the command-line program over a `DictionaryMapImpl`.

```
// Sigs2WordsMap.java - Problem 12.9: Sigs2WordsProto with a stored dictionary.
package predictive;

import java.io.IOException;
import java.util.TreeSet;

public class Sigs2WordsMap {

    public static void main(String[] args) throws IOException {
        Dictionary d = new DictionaryMapImpl(PredictivePrototype.DICTIONARY);   // the one
line that differs
        if (args.length == 0) {
            System.out.println("USAGE: java predictive.Sigs2WordsMap sig1 sig2 ... sigN");
            return;
        }
        for (String sig : args) {
            System.out.println(sig + ": " + new TreeSet<>(d.signatureToWords(sig)));
        }
    }
}
```

Listing 12.32 `Sigs2WordsTree.java`: the command-line program over a `DictionaryTreeImpl`.

```
// Sigs2WordsTree.java - Problem 12.9: Sigs2WordsProto with a stored dictionary.
package predictive;

import java.io.IOException;
import java.util.TreeSet;

public class Sigs2WordsTree {

    public static void main(String[] args) throws IOException {
        Dictionary d = new DictionaryTreeImpl(PredictivePrototype.DICTIONARY);   // the one
line that differs
        if (args.length == 0) {
            System.out.println("USAGE: java predictive.Sigs2WordsTree sig1 sig2 ... sigN");
            return;
        }
        for (String sig : args) {
            System.out.println(sig + ": " + new TreeSet<>(d.signatureToWords(sig)));
        }
    }
}
```

With `words.txt`, `java predictive.Sigs2WordsMap 4663 43556 96753 69 6263 47` prints the same six lines as the list and the prototype:

```
4663: [gone, good, goof, home, hone, hood, hoof]
43556: [hello]
96753: [world]
69: [my, ow, ox]
6263: [mane, name, oboe]
47: [is]
```

and `java predictive.Sigs2WordsTree` with the same signatures shows the prefixes:

```
4663: [gone, good, goof, home, hone, hood, hoof]
43556: [hello]
96753: [worke, world]
69: [my, ow, ox]
6263: [mane, name, oboe]
47: [ir, is]
```

The tree's answers differ in two lines, as they should: `worke` for 96753 (from `worked` and `worker`) and `ir` for 47 (from `iron` and `ironed`).

The Python solution. **Listing 12.33** and **Listing 12.34**; their output is identical to the Java output.

Listing 12.33 `sigs2words_map.py`: the map program in Python.

```
# sigs2words_map.py - Problem 12.9: sigs2words_proto with a stored dictionary.
import sys

from predictive.dictionary_map_impl import DictionaryMapImpl
from predictive.prototype import DICTIONARY


def main(args):
    d = DictionaryMapImpl(DICTIONARY)                  # the one line that differs
    if not args:
        print("USAGE: python -m predictive.sigs2words_map sig1 sig2 ... sigN")
        return
    for sig in args:
        print(sig + ": [" + ", ".join(sorted(d.signature_to_words(sig))) + "]")


if __name__ == "__main__":
    main(sys.argv[1:])
```

Listing 12.34 `sigs2words_tree.py`: the tree program in Python.

```
# sigs2words_tree.py - Problem 12.9: sigs2words_proto with a stored dictionary.
import sys

from predictive.dictionary_tree_impl import DictionaryTreeImpl
from predictive.prototype import DICTIONARY


def main(args):
    d = DictionaryTreeImpl(DICTIONARY)                  # the one line that differs
    if not args:
        print("USAGE: python -m predictive.sigs2words_tree sig1 sig2 ... sigN")
        return
    for sig in args:
        print(sig + ": [" + ", ".join(sorted(d.signature_to_words(sig))) + "]")


if __name__ == "__main__":
    main(sys.argv[1:])
```

```
4663: [gone, good, goof, home, hone, hood, hoof]
43556: [hello]
96753: [world]
69: [my, ow, ox]
6263: [mane, name, oboe]
47: [is]
4663: [gone, good, goof, home, hone, hood, hoof]
43556: [hello]
96753: [worke, world]
69: [my, ow, ox]
6263: [mane, name, oboe]
47: [ir, is]
```

The measurements. The timing tables of Problem 12.6 already cover all four dictionaries, and **Table 12.5** the four command-line programs. To see how each *scales*, we generated dictionaries of 25,000 to 400,000 words (`MakeDictionary` takes the count and the file name as arguments) and ran both timing programs on each. **Figure 12.4** plots time per lookup against size on logarithmic axes, where a line rising at 45 degrees means "proportional to n" and a flat line "independent of n".

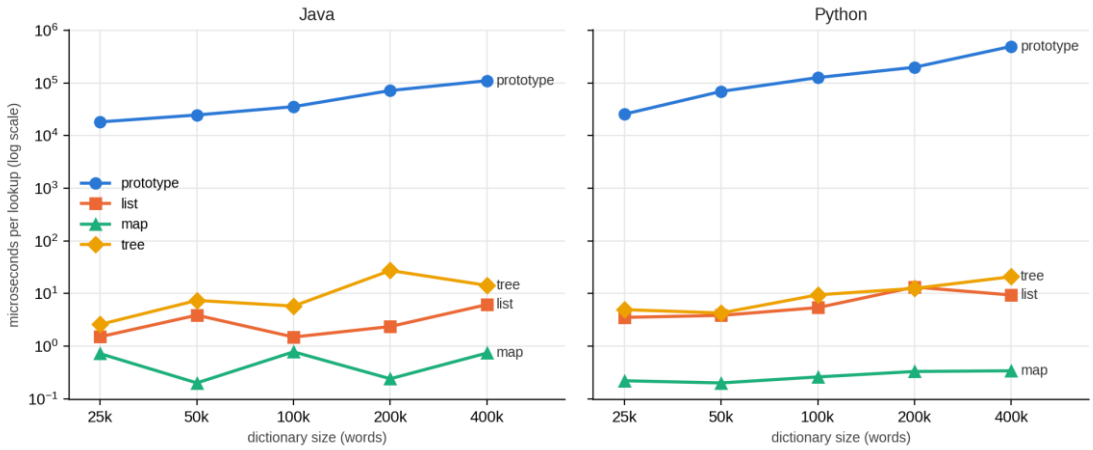


Figure 12.4 Microseconds per lookup against dictionary size, on logarithmic axes, measured with `TimingComparison` (left) and `timing_comparison.py` (right). The prototype rises with the size; the map stays flat; the list and the tree lie in between.

The picture confirms the formulas, with the noise of real measurements on a shared machine (the Java points jump as just-in-time compilation and garbage collection interfere). Sixteen times more words cost the prototype about six times more in Java and nineteen times more in Python, approaching Equation (12.3). The map stays flat below a microsecond, as Equation (12.8) promises. The list creeps up slowly, the log$n$ of Equation (12.6), with 15 to 19 comparisons that each touch a different part of memory. The tree creeps up too and lies *above* the list, as Equation (12.9) explains: for a whole signature it also collects and trims every longer word that begins with it, so it answers a bigger question, as its `found` column shows (7,496 against 5,332).

Table 12.7 gathers the whole comparison.

Table 12.7 The four dictionaries compared. n is the number of words, $\bar{m}$ their average length, k the number of matches, ℓ the length of the signature and W_s the number of words stored at its node.

Dictionary	Build	Lookup (whole signature)	Insert one word	Prefixes?	Ordered output?
Prototype	none	$O(n\bar{m})$, reads the file	edit the file	could, by scanning	could, by sorting
Sorted list	$O(n \log n)$	$O(\bar{m} \log n + k)$	$O(n)$ (shift)	yes, with a range search (Exercise 12.3)	yes
Hash map	$O(n\bar{m})$ expected	$O(\bar{m} + k)$ expected	$O(\bar{m})$ expected	no	no
Trie	$O\left(\sum_w \lvert w \rvert\right)$	$O(\ell + W_s \ell)$	$O(\lvert w \rvert)$	yes, built in	by walking the children in key order

So, to answer the problem: *yes*, list versus map is noticeable with many questions on a large dictionary (the map is more than ten times faster per lookup here, and builds faster), and tree versus map too, in the map's favor for whole signatures. But the tree answers a question the map cannot answer at all, which words the user could be typing *right now*, and that is why phones use tries and the GUI of Problem 12.10 uses the tree.

Line by line. The three Java programs differ only in the class after `new`. That is "programming to an interface": since `Dictionary` is the only type the rest of the code knows, the rest cannot depend on what differs between implementations. Python writes no declared type, but the three modules call only `signature_to_words`, the same discipline by convention.

A tempting wrong solution. The shortcut declares the concrete type, `DictionaryListImpl d = new DictionaryListImpl(...)`, and later someone calls `d.size()` because it is there. Now switching needs two changes, and the second does not compile for the map, whose method is `signatureCount()`. Declare variables with the interface type whenever the interface is enough.

> **CHECK YOURSELF.** In the Java table of Problem 12.6, the tree needed 13.57 µs per lookup and the map 0.22. If the phone asks the dictionary once per key press and a user presses four keys per second, does the difference matter?
>
> *Answer.* No: four lookups at 14 µs are 56 µs of work per second, far below anything a person notices. A factor of sixty matters for a program that asks millions of questions, not for a person who asks four per second; there, other qualities, such as the tree's prefixes, decide.

12.7 Part Four: A Keypad in MVC

12.7.1 Problem 12.10 — The predictive-text GUI

> **PROBLEM.** In a new package `predictivegui`, write a graphical user interface for the classes of Parts One to Three, using the Model–View–Controller method, so that the user interface is separate from the program code and can be changed without changing it. The *model* encapsulates a `Dictionary` object: it has a constructor that loads the dictionary and a method that queries it with signatures. It may store one dictionary and a list of the words entered so far, but it must not store signatures or spaces. The *controller* is the keypad: twelve buttons. The *view* is a text window above the keypad that shows the message; it must not be possible to type into it directly. Pressing key 2 gives one of a, b or c. Pressing `*` cycles through the words that match. Pressing `0` completes the current word and starts a new one after a space. Pressing `#` deletes the last letter of the current word. The GUI must work with any implementation of `Dictionary`, without changing the existing classes.

In our own words. We are building a small phone. Pressing 4, 6, 6, 3 shows `gone`; `*` replaces it with `good`, `goof`, then `home`; `0` fixes `home` and leaves a space; `#` takes back a letter. What must be remembered between presses belongs to the model; the buttons only report keys; the text window only shows what the model says.

Design. Step 4 of Section 12.2 first: *what must the model remember?* The finished words (`home`), the candidates for the current word (`gone`, `good`, `goof`, `home`), and which candidate is on show. Nothing else: the digits pressed need not be stored, because the word on show has one letter per key pressed, so its signature *is* the digits pressed. That is how we obey "no signatures in the model" without losing anything. **Table 12.8** lists what each key does.

Table 12.8 The model's three fields (finished words, candidates, index of the candidate on show) and what each key does to them. No signature is ever stored.

Key	Effect on the model
`2`–`9`	new candidates = the words for (signature of the word on show) + key; if there are none, the key is ignored
`*`	show the next candidate; after the last one, the first again
`0`	append the word on show to the finished words; no candidates any more
`#`	new candidates = the words for the signature of the word on show without its last digit
`1`, others	nothing

A key that matches nothing is ignored: showing digits instead would force the model to store them, and a phone that refuses an impossible key behaves like a phone. With the tree dictionary, a

key is refused only when no word at all begins with the keys pressed; after 4, 6, 6 the candidates are the prefixes `gon`, `goo`, `hom`, `hon`, `hoo` and `inn`.

The rest is the MVC structure of Chapter 11 [10], drawn in **Figure 12.5**. A user action on the view's buttons reaches the controller; the controller calls `model.press(key)`; the model changes and notifies every registered listener; the view, one of those listeners, asks the model for the text and shows it.

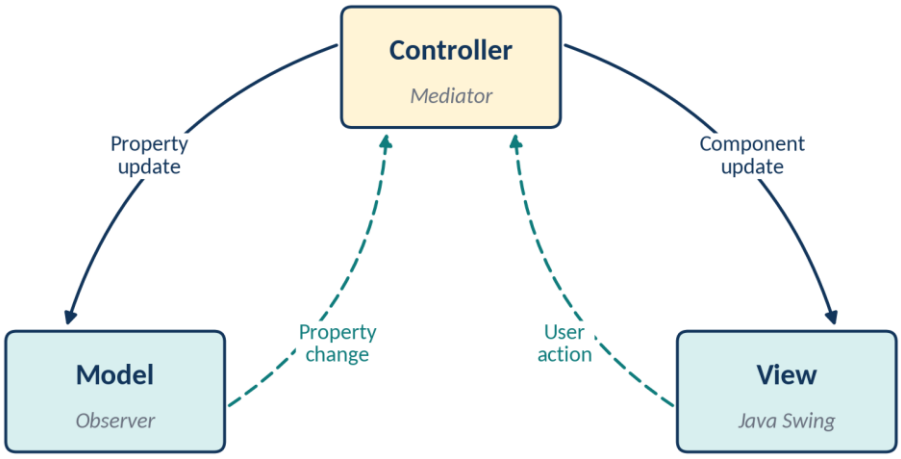


Figure 12.5 The Model–View–Controller pattern. The controller mediates: it turns user actions into updates of the model, and the model's changes reach the view.

For notification, the model keeps a `List<Runnable>` of listeners and runs each after every change. Older code, including the first version of this program, extends `java.util.Observable` instead, which is *deprecated* since Java 9 ("still there, do not use it in new code") because it forces the model to extend a class and tells listeners nothing about what changed [5].

Figure 12.6 shows that first version's window, and a warning: key 9 is labeled `vwxy`, while **Figure 12.1** puts w, x, y and z on 9 and v on 8. The label was typed by hand in the controller, apart from the letters the signature code uses, and no test compared them. Our controller takes its labels from one table next to the keys, matching `keyOf`.



Figure 12.6 The window of an earlier version of this program, rebuilt with its hand-typed key labels, before any key is pressed. The label of key 9 is wrong: it reads `vwxy` instead of `wxyz`.

The Java solution. Four classes, one per role, plus a test. The model, **Listing 12.35**, depends on the `Dictionary` interface only; its second constructor is a convenience that builds the tree.

Listing 12.35 `DictionaryModel.java`: the model. It stores the finished words, the candidates and an index, and recomputes the signature from the word on show.

```
// DictionaryModel.java - Problem 12.10: the model of the keypad GUI. It holds the
// dictionary, the finished words and the candidates for the word being typed.
package predictivegui;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;

import predictive.Dictionary;
import predictive.DictionaryTreeImpl;
import predictive.PredictivePrototype;

public class DictionaryModel {

    private final Dictionary dictionary;
    private final List<String> message = new ArrayList<>();      // finished words only
    private List<String> candidates = new ArrayList<>();         // matches for the current
word
    private int index = 0;                                        // the candidate on show
    private final List<Runnable> listeners = new ArrayList<>();

    /** A model over any Dictionary: tests pass a small fake one. */
    public DictionaryModel(Dictionary dictionary) {
        this.dictionary = dictionary;
    }

    /** The model of the real program: a prefix-matching tree read from a file. */
    public DictionaryModel(String path) throws IOException {
        this(new DictionaryTreeImpl(path));
    }

    /** Registers code to run after every change (the views' update methods). */
    public void addListener(Runnable listener) {
        listeners.add(listener);
    }

    /** Queries the dictionary: the matches of a signature, in alphabetical order. */
    public List<String> lookUp(String signature) {
        Set<String> found = dictionary.signatureToWords(signature);
        return new ArrayList<>(new TreeSet<>(found));
    }
```

Listing 12.35 (Continued) *DictionaryModel.java: the model. It stores the finished words, the candidates and an index, and recomputes the signature from the word on show.*

```
    /** One key press: '2'..'9' type, '*' next match, '0' finish the word, '#' delete. */
    public void press(char key) {
        switch (key) {
            case '0' -> finishWord();
            case '*' -> nextCandidate();
            case '#' -> deleteLetter();
            default -> typeKey(key);
        }
        for (Runnable listener : listeners) {
            listener.run();                          // every view redraws itself
        }
    }

    private String currentWord() {
        return candidates.isEmpty() ? "" : candidates.get(index);
    }

    private void typeKey(char key) {
        if (key < '2' || key > '9') {
            return;                                  // '1' and unknown keys do nothing
        }
        // the signature is never stored: it is recomputed from the word on show
        List<String> found = lookUp(PredictivePrototype.wordToSignature(currentWord()) +
key);
        if (!found.isEmpty()) {                      // a key that matches nothing is
ignored
            candidates = found;
            index = 0;
        }
    }

    private void nextCandidate() {
        if (!candidates.isEmpty()) {
            index = (index + 1) % candidates.size(); // after the last one, back to the
first
        }
    }

    private void finishWord() {
        if (!candidates.isEmpty()) {
            message.add(currentWord());
            candidates = new ArrayList<>();
            index = 0;
        }
    }
```

Listing 12.35 (Continued) *DictionaryModel.java: the model. It stores the finished words, the candidates and an index, and recomputes the signature from the word on show.*

```
    private void deleteLetter() {
        String word = currentWord();
        if (word.isEmpty()) {
            return;
        }
        String shorter = PredictivePrototype.wordToSignature(word).substring(0,
word.length() - 1);
        candidates = shorter.isEmpty() ? new ArrayList<>() : lookUp(shorter);
        index = 0;
    }

    /** The finished words followed by the current word, if there is one. */
    public List<String> getWords() {
        List<String> all = new ArrayList<>(message);
        if (!candidates.isEmpty()) {
            all.add(currentWord());
        }
        return Collections.unmodifiableList(all);
    }

    /** The text to display: words separated by spaces; a finished word ends with one. */
    public String getText() {
        StringBuilder sb = new StringBuilder(String.join(" ", message));
        if (!message.isEmpty()) {
            sb.append(' ');
        }
        return sb.append(currentWord()).toString();
    }
}
```

The view, **Listing 12.36**, is a panel with a text area that the user cannot edit; `update()` copies the model's text into it.

Listing 12.36 `MessageView.java`: the view, a read-only text area inside a scroll pane.

```
// MessageView.java - Problem 12.10: the view. It shows the model's text and
// cannot be edited by typing.
package predictivegui;

import java.awt.BorderLayout;

import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public final class MessageView extends JPanel {

    private final DictionaryModel model;
    private final JTextArea message = new JTextArea(5, 20);

    public MessageView(DictionaryModel model) {
        super(new BorderLayout());
        this.model = model;
        message.setEditable(false);                  // only the keypad may change the text
        message.setLineWrap(true);
        message.setWrapStyleWord(true);
        message.setText("predictive text: enter text with 8 keys.");
        add(new JScrollPane(message), BorderLayout.CENTER);
    }

    /** Called by the model after every change. */
    public void update() {
        message.setText(model.getText());
    }

    /** What the view shows now (used by the self-test). */
    public String getText() {
        return message.getText();
    }
}
```

The controller, **Listing 12.37**, builds the twelve buttons from a table and gives each a listener that calls `model.press` with its own key.

Listing 12.37 `KeypadController.java`: the controller, twelve buttons in a 4 by 3 grid built from one table of keys and labels.

```
// KeypadController.java - Problem 12.10: the controller. Twelve buttons in a
// 4 x 3 grid; each button passes its key to the model and does nothing else.
package predictivegui;

import java.awt.GridLayout;
import java.util.HashMap;
import java.util.Map;

import javax.swing.JButton;
import javax.swing.JPanel;

public final class KeypadController extends JPanel {

    private static final String[][] KEYS = {
        {"1", ""}, {"2", "abc"}, {"3", "def"},
        {"4", "ghi"}, {"5", "jkl"}, {"6", "mno"},
        {"7", "pqrs"}, {"8", "tuv"}, {"9", "wxyz"},
        {"*", "next"}, {"0", "space"}, {"#", "delete"}};

    private final Map<Character, JButton> buttons = new HashMap<>();

    public KeypadController(DictionaryModel model) {
        super(new GridLayout(4, 3));
        for (String[] k : KEYS) {
            char key = k[0].charAt(0);              // a new variable in every turn of the
loop
            JButton button = new JButton("<html><center>" + k[0] + "<br>" + k[1] +
"</center></html>");
            button.addActionListener(e -> model.press(key));
            button.setEnabled(key != '1');          // key 1 carries no letters
            buttons.put(key, button);
            add(button);
        }
    }

    /** The button of a key (used by the self-test to click it). */
    public JButton buttonFor(char key) {
        return buttons.get(key);
    }
}
```

Listing 12.38 puts them together. Without arguments it opens the window; with `--selftest` it builds the same components without a window, clicks the buttons of the example from code, and prints what the view shows after each click.

Listing 12.38 `PredictiveGUI.java`: the main program, with a `--selftest` mode that clicks the buttons from code.

```java
// PredictiveGUI.java - Problem 12.10: puts model, view and controller together.
// Run with --selftest to click the buttons from code, without opening a window.
package predictivegui;

import java.awt.BorderLayout;
import java.io.IOException;

import javax.swing.JFrame;
import javax.swing.SwingUtilities;

import predictive.PredictivePrototype;

public class PredictiveGUI {

    public static void main(String[] args) throws IOException {
        DictionaryModel model = new DictionaryModel(PredictivePrototype.DICTIONARY);
        if (args.length > 0 && args[0].equals("--selftest")) {
            MessageView view = new MessageView(model);
            KeypadController keypad = new KeypadController(model);
            model.addListener(view::update);
            selfTest(view, keypad);
            return;
        }
        SwingUtilities.invokeLater(() -> {           // Swing objects live on the event
thread
            MessageView view = new MessageView(model);
            KeypadController keypad = new KeypadController(model);
            model.addListener(view::update);
            JFrame frame = new JFrame("Predictive Text");
            frame.add(view, BorderLayout.CENTER);
            frame.add(keypad, BorderLayout.SOUTH);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(260, 340);
            frame.setVisible(true);
        });
    }

    /** Clicks the buttons of the book's example and prints what the view shows. */
    static void selfTest(MessageView view, KeypadController keypad) {
        String presses = "4663***0" + "96753#0";
        System.out.println("start      -> \"" + view.getText() + "\"");
        for (char key : presses.toCharArray()) {
            keypad.buttonFor(key).doClick();         // exactly what a mouse click does
            System.out.println("press " + key + "    -> \"" + view.getText() + "\"");
        }
    }
}
```

`java predictivegui.PredictiveGUI --selftest`, run from the `java` folder, prints:

```
start      -> "predictive text: enter text with 8 keys."
press 4    -> "g"
press 6    -> "go"
press 6    -> "gon"
press 3    -> "gone"
press *    -> "good"
press *    -> "goof"
press *    -> "home"
press 0    -> "home "
press 9    -> "home w"
press 6    -> "home wo"
press 7    -> "home wor"
press 5    -> "home work"
press 3    -> "home worke"
press #    -> "home work"
press 0    -> "home work "
```

Read the output as a film of the screen: `g` after 4; `gon`, first of six prefixes, after 4, 6, 6; the words of 4663 after the 3, walked through with `*` up to `home`; `0` fixes it. In the second word, `3` gives `worke` (from `worked`), `#` takes back the `e` and the model looks up 9675 afresh, whose first candidate is `work`.

Listing 12.39 tests the model alone. Since `Dictionary` has one method, a lambda over a small `Map` is a complete fake dictionary, and the test needs no file.

Listing 12.39 `ModelCheck.java`: tests of the model alone, with a fake dictionary written as a lambda.

```
// ModelCheck.java - Problem 12.10: tests for the model alone, with a fake
// dictionary written as a lambda. No window, no buttons.
package predictivegui;

import java.util.Map;
import java.util.Set;

import predictive.Dictionary;

public class ModelCheck {

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    private static void type(DictionaryModel m, String keys) {
        for (char k : keys.toCharArray()) {
            m.press(k);
        }
    }

    public static void main(String[] args) {
        Map<String, Set<String>> table = Map.of(
                "4", Set.of("g", "h", "i"), "43", Set.of("he", "if"),
                "435", Set.of("hel"), "4355", Set.of("hell"), "43556", Set.of("hello"),
                "9", Set.of("w"), "96", Set.of("wo", "yo"));
        Dictionary fake = sig -> table.getOrDefault(sig, Set.of());   // Dictionary has one
method

        DictionaryModel m = new DictionaryModel(fake);
        int[] updates = {0};
        m.addListener(() -> updates[0]++);
```

Listing 12.39 (Continued) *ModelCheck.java: tests of the model alone, with a fake dictionary written as a lambda.*

```
        type(m, "43");
        check("first match", "he", m.getText());
        m.press('*');
        check("star cycles", "if", m.getText());
        m.press('*');
        check("star wraps", "he", m.getText());
        type(m, "556");
        check("hello", "hello", m.getText());
        m.press('7');
        check("no match: key ignored", "hello", m.getText());
        m.press('0');
        check("0 finishes a word", "hello ", m.getText());
        check("words", "[hello]", m.getWords().toString());
        m.press('0');
        check("0 again adds nothing", "[hello]", m.getWords().toString());
        type(m, "96*");
        check("second word", "hello yo", m.getText());
        type(m, "##");
        check("delete back to nothing", "hello ", m.getText());
        m.press('#');
        check("delete on empty", "hello ", m.getText());
        check("one update per press", 16, updates[0]);
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
PASS first match
PASS star cycles
PASS star wraps
PASS hello
PASS no match: key ignored
PASS 0 finishes a word
PASS words
PASS 0 again adds nothing
PASS second word
PASS delete back to nothing
PASS delete on empty
PASS one update per press
All 12 checks passed.
```

Figure 12.7 shows the real window, opened on a virtual screen at three moments.

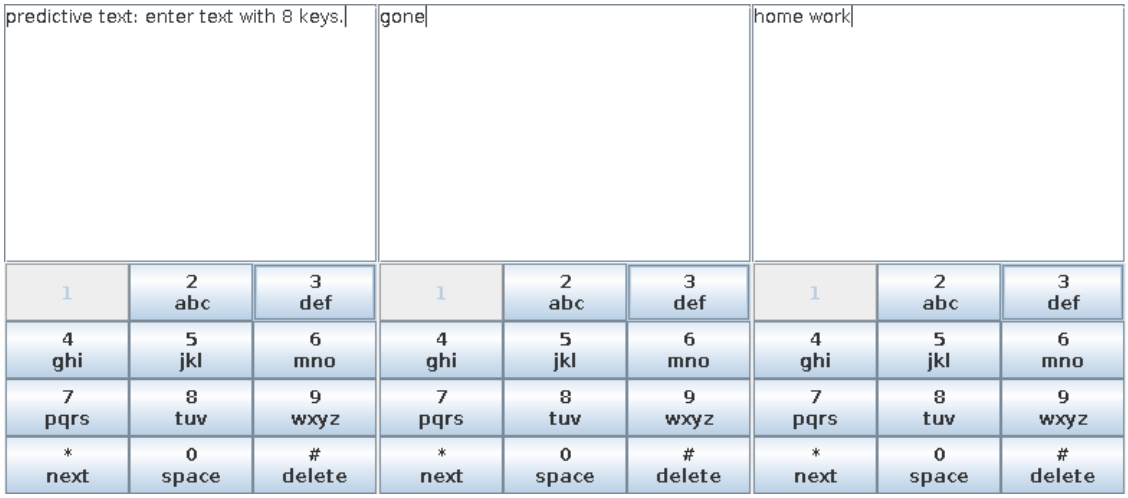


Figure 12.7 The Swing window of **Listing 12.38** at the start (left), after pressing 4, 6, 6, 3 (middle), and after `* * * 0 9 6 7 5 3 #` (right). Key 1 is disabled because it carries no letters.

The Python solution. The same four roles. The model, **Listing 12.40**, imports nothing from Tkinter; the view and the controller import `tkinter` [11] *inside* their constructors, so the model and the self-test also run where Tkinter is not installed (some Linux systems ship Python without it). **Listing 12.41** is the view, **Listing 12.42** the controller, and **Listing 12.44** tests the model with a fake dictionary.

Listing 12.40 `dictionary_model.py`: the model in Python; `from_file` is an alternative constructor, written as a class method.

```
# dictionary_model.py - Problem 12.10: the model of the keypad GUI. It holds the
# dictionary, the finished words and the candidates for the word being typed.
from predictive.dictionary_tree_impl import DictionaryTreeImpl
from predictive.prototype import word_to_signature


class DictionaryModel:

    def __init__(self, dictionary):
        """A model over any Dictionary: tests pass a small fake one."""
        self._dictionary = dictionary
        self._message = []            # finished words only
        self._candidates = []         # matches for the current word
        self._index = 0               # the candidate on show
        self._listeners = []

    @classmethod
    def from_file(cls, path):
        """The model of the real program: a prefix-matching tree read from a file."""
        return cls(DictionaryTreeImpl(path))

    def add_listener(self, listener):
        """Register a function to call after every change (the views' update)."""
        self._listeners.append(listener)

    def look_up(self, signature):
        """Query the dictionary: the matches of a signature, in alphabetical order."""
        return sorted(self._dictionary.signature_to_words(signature))

    def press(self, key):
        """One key press: '2'..'9' type, '*' next match, '0' finish the word, '#'
delete."""
        if key == "0":
            self._finish_word()
        elif key == "*":
            self._next_candidate()
        elif key == "#":
            self._delete_letter()
        else:
            self._type_key(key)
        for listener in self._listeners:
            listener()                                # every view redraws itself

    def _current_word(self):
        return self._candidates[self._index] if self._candidates else ""
```

Listing 12.40 (Continued) *dictionary_model.py: the model in Python; from_file is an alternative constructor, written as a class method.*

```
    def _type_key(self, key):
        if not "2" <= key <= "9":
            return                                  # '1' and unknown keys do nothing
        # the signature is never stored: it is recomputed from the word on show
        found = self.look_up(word_to_signature(self._current_word()) + key)
        if found:                                   # a key that matches nothing is
ignored
            self._candidates, self._index = found, 0

    def _next_candidate(self):
        if self._candidates:
            self._index = (self._index + 1) % len(self._candidates)

    def _finish_word(self):
        if self._candidates:
            self._message.append(self._current_word())
            self._candidates, self._index = [], 0

    def _delete_letter(self):
        word = self._current_word()
        if not word:
            return
        shorter = word_to_signature(word)[:-1]
        self._candidates = self.look_up(shorter) if shorter else []
        self._index = 0

    def get_words(self):
        """The finished words followed by the current word, if there is one."""
        words = list(self._message)
        if self._candidates:
            words.append(self._current_word())
        return tuple(words)                         # a tuple cannot be changed by callers

    def get_text(self):
        """The text to display: words separated by spaces; a finished word ends with
one."""
        text = " ".join(self._message)
        if self._message:
            text += " "
        return text + self._current_word()
```

Listing 12.41 `message_view.py`: the view, a Tkinter `Text` widget that is unlocked only while the model's text is written into it.

```
# message_view.py - Problem 12.10: the view. It shows the model's text and
# cannot be edited by typing.


class MessageView:

    def __init__(self, parent, model):
        import tkinter as tk                          # imported only when a window is built
        self._model = model
        self._text = tk.Text(parent, width=24, height=6, wrap="word")
        self._text.pack(fill="both", expand=True)
        self._show("predictive text: enter text with 8 keys.")

    def _show(self, text):
        self._text.configure(state="normal")          # unlock, replace, lock again
        self._text.delete("1.0", "end")
        self._text.insert("1.0", text)
        self._text.configure(state="disabled")        # only the keypad may change the text

    def update(self):
        """Called by the model after every change."""
        self._show(self._model.get_text())

    def get_text(self):
        return self._text.get("1.0", "end-1c")
```

Listing 12.42 `keypad_controller.py`: the controller; `make_command` gives every button a function that remembers its own key.

```
# keypad_controller.py - Problem 12.10: the controller. Twelve buttons in a
# 4 x 3 grid; each button passes its key to the model and does nothing else.

KEYS = [("1", ""), ("2", "abc"), ("3", "def"),
        ("4", "ghi"), ("5", "jkl"), ("6", "mno"),
        ("7", "pqrs"), ("8", "tuv"), ("9", "wxyz"),
        ("*", "next"), ("0", "space"), ("#", "delete")]


def make_command(model, key):
    """The function a button calls. Each call of make_command has its own 'key'."""
    return lambda: model.press(key)


class KeypadController:

    def __init__(self, parent, model):
        import tkinter as tk                          # imported only when a window is built
        frame = tk.Frame(parent)
        frame.pack(fill="both")
        self._buttons = {}
        for n, (key, letters) in enumerate(KEYS):
            button = tk.Button(frame, text=key + "\n" + letters, width=6,
                               command=make_command(model, key))
            if key == "1":
                button.configure(state="disabled")  # key 1 carries no letters
            button.grid(row=n // 3, column=n % 3, sticky="nsew")
            self._buttons[key] = button

    def button_for(self, key):
        return self._buttons[key]
```

Listing 12.43 `predictive_gui.py`: the main program; `--selftest` presses the keys through the same `make_command` functions that the buttons use.

```
# predictive_gui.py - Problem 12.10: puts model, view and controller together.
# Run with --selftest to press the keys from code, without opening a window.
import sys

from predictive.prototype import DICTIONARY
from predictivegui.dictionary_model import DictionaryModel
from predictivegui.keypad_controller import make_command


class TextView:
    """A stand-in view for the self-test: it keeps the text instead of drawing it."""

    def __init__(self, model):
        self._model = model
        self.text = "predictive text: enter text with 8 keys."

    def update(self):
        self.text = self._model.get_text()


def self_test(model):
    view = TextView(model)
    model.add_listener(view.update)
    print('start      -> "' + view.text + '"')
    for key in "4663***0" + "96753#0":
        make_command(model, key)()                  # exactly what a button click calls
        print("press " + key + '    -> "' + view.text + '"')


def main(args):
    model = DictionaryModel.from_file(DICTIONARY)
    if args and args[0] == "--selftest":
        self_test(model)
        return
    import tkinter as tk
    from predictivegui.keypad_controller import KeypadController
    from predictivegui.message_view import MessageView
    root = tk.Tk()
    root.title("Predictive Text")
    view = MessageView(root, model)
    KeypadController(root, model)
    model.add_listener(view.update)
    root.mainloop()
```

Listing 12.43 (Continued) *predictive_gui.py: the main program; --selftest presses the keys through the same make_command functions that the buttons use.*

```
if __name__ == "__main__":
    main(sys.argv[1:])
```

`python -m predictivegui.predictive_gui --selftest` prints the same film as the Java version:

```
start      -> "predictive text: enter text with 8 keys."
press 4    -> "g"
press 6    -> "go"
press 6    -> "gon"
press 3    -> "gone"
press *    -> "good"
press *    -> "goof"
press *    -> "home"
press 0    -> "home "
press 9    -> "home w"
press 6    -> "home wo"
press 7    -> "home wor"
press 5    -> "home work"
press 3    -> "home worke"
press #    -> "home work"
press 0    -> "home work "
```

Listing 12.44 `model_check.py`: tests of the model alone; the fake dictionary is any object with a `signature_to_words` method.

```
# model_check.py - Problem 12.10: tests for the model alone, with a fake
# dictionary. No window, no buttons.
import unittest

from predictivegui.dictionary_model import DictionaryModel

TABLE = {"4": {"g", "h", "i"}, "43": {"he", "if"}, "435": {"hel"}, "4355": {"hell"},
         "43556": {"hello"}, "9": {"w"}, "96": {"wo", "yo"}}


class FakeDictionary:
    """Anything with signature_to_words() will do: duck typing."""

    def signature_to_words(self, signature):
        return TABLE.get(signature, set())


class TestModel(unittest.TestCase):
    def setUp(self):
        self.m = DictionaryModel(FakeDictionary())
        self.updates = 0

        def count():
            self.updates += 1
        self.m.add_listener(count)

    def type(self, keys):
        for k in keys:
            self.m.press(k)

    def test_typing_and_cycling(self):
        self.type("43")
        self.assertEqual("he", self.m.get_text())
        self.type("*")
        self.assertEqual("if", self.m.get_text())
        self.type("*")
        self.assertEqual("he", self.m.get_text())            # wraps around
        self.type("556")
        self.assertEqual("hello", self.m.get_text())
        self.type("7")
        self.assertEqual("hello", self.m.get_text())         # no match: key ignored
```

Listing 12.44 (Continued) *model_check.py: tests of the model alone; the fake dictionary is any object with a signature_to_words method.*

```
    def test_finishing_and_deleting(self):
        self.type("435560")
        self.assertEqual("hello ", self.m.get_text())
        self.assertEqual(("hello",), self.m.get_words())
        self.type("0")
        self.assertEqual(("hello",), self.m.get_words())     # 0 again adds nothing
        self.type("96*")
        self.assertEqual("hello yo", self.m.get_text())
        self.type("###")
        self.assertEqual("hello ", self.m.get_text())         # delete stops at nothing

    def test_one_update_per_press(self):
        self.type("43*0#")
        self.assertEqual(5, self.updates)


if __name__ == "__main__":
    unittest.main(verbosity=2)
```

```
test_finishing_and_deleting (__main__.TestModel.test_finishing_and_deleting) ... ok
test_one_update_per_press (__main__.TestModel.test_one_update_per_press) ... ok
test_typing_and_cycling (__main__.TestModel.test_typing_and_cycling) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Figure 12.8 shows the Tkinter window at the same three moments.



Figure 12.8 The Tkinter window of **Listing 12.43** at the same three moments as in **Figure 12.7**.

Line by line. In **Listing 12.35**, the fields match **Table 12.8**. `press` switches to one small private method per key and then runs every listener, changed or not; the test `one update per press` pins that down (Exercise 12.8 improves it). In `typeKey`, `wordToSignature(currentWord()) + key` is "the digits so far, plus the new one". `lookUp` sorts the answer through a `TreeSet` and copies it into an `ArrayList`, because the model needs positions. `nextCandidate` wraps with `%`, and `deleteLetter` drops the last digit with `substring` and looks up the shorter signature, emptying the candidates when nothing is left. `getWords` returns an unmodifiable list.

In **Listing 12.36**, `setEditable(false)` makes the model the only source of text. In **Listing 12.37**, `char key = k[0].charAt(0);` creates a *new* variable in every turn, which the lambda `e -> model.press(key)` may capture because it is never reassigned (Problem 12.16 is about this line). HTML labels give each button two centered lines, and `buttonFor` lets the self-test call `doClick()`,

which does exactly what a mouse click does. In **Listing 12.38**, the window is built inside `SwingUtilities.invokeLater(...)`, because Swing components belong to the *event dispatch thread* [12]; the self-test, with no window and no event thread, builds them on the main thread. The view is registered by the main program, not by its own constructor: an object that hands out `this` before its constructor ends can be called half-built.

In Python, `from_file` is a `@classmethod`, the usual way to write a second constructor, and `get_words` returns a `tuple`. The Tkinter `Text` widget has no "read-only but writable by the program" mode, so `_show` unlocks it, replaces the text and locks it again. `grid(row=n // 3, column=n % 3)` lays twelve buttons out as four rows of three, and `make_command(model, key)` gives each its own function (Problem 12.16 explains why).

A tempting wrong solution. The first temptation is to let the text area *be* the model: read the message back with `getText()`, change it, write it with `setText()`. Then `*` must guess the candidates from the text, `#` must guess where the word begins, the area must stay editable, so the user can corrupt the only copy of the data, and nothing can be tested without a window. The second temptation stores `currentSig` next to the words, as the old design did. Besides breaking the rule, two fields describing one fact can disagree, after a `#` that updates only one of them, for example; one field recomputed on demand cannot.

> **CHECK YOURSELF.** Starting from an empty model with the tree dictionary of `words.txt`, what does the view show after the presses `4 3 5 5 6 0 # 9 6`?
>
> *Answer.* `hello wo`. The keys 4, 3, 5, 5, 6 give `hello` (the only candidate for 43556), `0` fixes it, and the `#` does nothing, because there is no current word to delete from (the finished words are not edited). Then 9 gives `w` and 6 gives `wo`, the first of the candidates for 96 in alphabetical order.

12.8 New Worked Problems: Collections and Generics

The predictive-text project exercised lists, sets, maps, `Comparable`, interfaces and a GUI. The next three problems take three ideas of Chapter 9 further, each in the Profitina Laundry shop: choosing a collection by measuring it, writing a generic class of our own, and ordering objects by several keys at once.

12.8.1 Problem 12.11 — Choosing a collection by measuring it

> **PROBLEM.** At the pickup counter of Profitina Laundry, every order has a number, and the shop keeps the numbers of the orders that are ready (100,000 of them on a busy day at a large branch). A scanner reads the number on a customer's slip, and the program must say at once whether that order is ready. At the end of the day, the numbers must be printed in increasing order; and when the scanner reads the same slip several times, the log of scans should keep each number once, in the order of its first scan. Choose a collection for each of the three jobs, justify the choice, and confirm it with measurements.

In our own words. Three jobs, three questions. *Is this number present?*, asked thousands of times. *All numbers in order*, asked once a day. *Remove repeats but keep the first-seen order*, asked for a log of scans.

Design. Step 2 of Section 12.2 matches each job to a structure (**Table 12.9**); the program then measures.

Table 12.9 Three jobs of the pickup counter and the collections that could do them. The costs are per operation unless stated otherwise.

Job	Candidate	Cost of the job	Verdict
"Is x present?"	`ArrayList` / `list`	$O(n)$: looks at every element	too slow for many questions
"Is x present?"	`HashSet` / `set`	$O(1)$ expected	the choice
"Is x present?"	`TreeSet` / sorted list + `bisect`	$O(\log n)$	fine, and ordered as well
"All in order"	`TreeSet` / `sorted(...)`	iteration in order, $O(n)$	the choice
"Unique, first-seen order"	`LinkedHashSet` / `dict.fromkeys`	$O(1)$ expected per item	the choice
"Unique, first-seen order"	`ArrayList` + `contains`	$O(n)$ per item, $O(n^2)$ in total	the tempting wrong solution

Python has no sorted set; a sorted list searched with `bisect`, or `sorted(the_set)` for a listing, replaces it. And since Python 3.7 a `dict` remembers insertion order, so `list(dict.fromkeys(items))` removes repeats in first-seen order.

The Java solution. Listing 12.45 builds 100,000 even order numbers in random order, asks 10,000 questions (about half about odd, absent numbers), and times each candidate; it also prints the iteration order of three sets and times two ways of removing repeated scans.

Listing 12.45 `CollectionChoice.java`: membership tests on three collections, the iteration order of three sets, and two ways of removing duplicates, all timed.

```
// CollectionChoice.java - Problem 12.11: choosing a collection by what it must do,
// then measuring. Profitina Laundry checks scanned order numbers.
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Random;
import java.util.Set;
import java.util.TreeSet;
import java.util.function.IntPredicate;

public class CollectionChoice {

    /** Milliseconds for all the membership tests; also counts the hits. */
    static double timeContains(IntPredicate contains, int[] queries, int[] hits) {
        long t0 = System.nanoTime();
        int h = 0;
        for (int q : queries) {
            if (contains.test(q)) {
                h++;
            }
        }
        hits[0] = h;
        return (System.nanoTime() - t0) / 1e6;
    }

    /** Tempting but wrong: remove duplicates with List.contains, O(n) per call. */
    static List<Integer> dedupWithList(int[] ids) {
        List<Integer> seen = new ArrayList<>();
        for (int id : ids) {
            if (!seen.contains(id)) {
                seen.add(id);
            }
        }
        return seen;
    }
```

Listing 12.45 (Continued) *CollectionChoice.java: membership tests on three collections, the iteration order of three sets, and two ways of removing duplicates, all timed.*

```
    /** The same result with a LinkedHashSet: O(1) expected per call, order kept. */
    static List<Integer> dedupWithSet(int[] ids) {
        Set<Integer> seen = new LinkedHashSet<>();
        for (int id : ids) {
            seen.add(id);
        }
        return new ArrayList<>(seen);
    }

    public static void main(String[] args) {
        Random random = new Random(42);                // a fixed seed: the same ids every
run
        int n = 100_000;
        List<Integer> ids = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            ids.add(100_000 + 2 * i);                    // even order numbers only
        }
        Collections.shuffle(ids, random);
        int[] queries = new int[10_000];
        for (int k = 0; k < queries.length; k++) {
            queries[k] = 100_000 + random.nextInt(2 * n);   // about half are odd: absent
        }

        List<Integer> list = new ArrayList<>(ids);
        Set<Integer> hash = new HashSet<>(ids);
        Set<Integer> tree = new TreeSet<>(ids);
        int[] hits = new int[1];
        System.out.println("contains: " + n + " ids, " + queries.length + " queries");
        System.out.printf("  ArrayList %9.2f ms%n", timeContains(list::contains, queries,
hits));
        int listHits = hits[0];
        System.out.printf("  HashSet   %9.2f ms%n", timeContains(hash::contains, queries,
hits));
        System.out.printf("  TreeSet   %9.2f ms%n", timeContains(tree::contains, queries,
hits));
        System.out.println("  same hits: " + (listHits == hits[0]) + " (" + hits[0] + ")");

        List<Integer> few = List.of(42, 7, 19, 7, 88, 3);
        System.out.println("order of " + few + ":");
        System.out.println("  HashSet       " + new HashSet<>(few));
        System.out.println("  LinkedHashSet " + new LinkedHashSet<>(few));
        System.out.println("  TreeSet       " + new TreeSet<>(few));
```

Listing 12.45 (Continued) *CollectionChoice.java: membership tests on three collections, the iteration order of three sets, and two ways of removing duplicates, all timed.*

```
        for (int m : new int[] {20_000, 40_000, 80_000}) {
            int[] scans = new int[m];
            for (int k = 0; k < m; k++) {
                scans[k] = random.nextInt(m / 2);        // many repeated scans
            }
            long t0 = System.nanoTime();
            List<Integer> a = dedupWithList(scans);
            long t1 = System.nanoTime();
            List<Integer> b = dedupWithSet(scans);
            long t2 = System.nanoTime();
            System.out.printf("dedup %6d scans: List %8.2f ms, LinkedHashSet %5.2f ms,
equal: %b%n",
                    m, (t1 - t0) / 1e6, (t2 - t1) / 1e6, a.equals(b));
        }
    }
}
```

```
contains: 100000 ids, 10000 queries
  ArrayList   1469.95 ms
  HashSet        5.45 ms
  TreeSet       15.75 ms
  same hits: true (4999)
order of [42, 7, 19, 7, 88, 3]:
  HashSet       [19, 3, 7, 88, 42]
  LinkedHashSet [42, 7, 19, 88, 3]
  TreeSet       [3, 7, 19, 42, 88]
dedup  20000 scans: List    85.85 ms, LinkedHashSet  7.73 ms, equal: true
dedup  40000 scans: List   278.55 ms, LinkedHashSet  8.15 ms, equal: true
dedup  80000 scans: List   964.64 ms, LinkedHashSet  6.37 ms, equal: true
```

The Python solution. **Listing 12.46** runs the same experiment with `list`, `set` and a sorted list searched with `bisect`.

Listing 12.46 `collection_choice.py`: the same experiment in Python; a sorted list with `bisect` stands in for `TreeSet`.

```
# collection_choice.py - Problem 12.11: choosing a collection by what it must do,
# then measuring. Profitina Laundry checks scanned order numbers.
import random
import time
from bisect import bisect_left


def time_contains(contains, queries):
    """Milliseconds for all the membership tests, and the number of hits."""
    t0 = time.perf_counter()
    hits = sum(1 for q in queries if contains(q))
    return (time.perf_counter() - t0) * 1e3, hits


def in_sorted(sorted_list, x):
    """Binary search in a sorted list: Python's stand-in for a TreeSet."""
    i = bisect_left(sorted_list, x)
    return i < len(sorted_list) and sorted_list[i] == x


def dedup_with_list(ids):
    """Tempting but wrong: remove duplicates with 'in' on a list, O(n) per test."""
    seen = []
    for i in ids:
        if i not in seen:
            seen.append(i)
    return seen


def dedup_with_dict(ids):
    """The same result with dict keys: O(1) expected per test, order kept."""
    return list(dict.fromkeys(ids))


def main():
    rng = random.Random(42)                      # a fixed seed: the same ids every run
    n = 100_000
    ids = [100_000 + 2 * i for i in range(n)]    # even order numbers only
    rng.shuffle(ids)
    queries = [100_000 + rng.randrange(2 * n) for _ in range(10_000)]
```

Listing 12.46 (Continued) *collection_choice.py: the same experiment in Python; a sorted list with bisect stands in for TreeSet.*

```
    as_list, as_set, as_sorted = list(ids), set(ids), sorted(ids)
    print(f"contains: {n} ids, {len(queries)} queries")
    ms, list_hits = time_contains(as_list.__contains__, queries)
    print(f"  list      {ms:9.2f} ms")
    ms, hits = time_contains(as_set.__contains__, queries)
    print(f"  set       {ms:9.2f} ms")
    ms, hits = time_contains(lambda q: in_sorted(as_sorted, q), queries)
    print(f"  sorted    {ms:9.2f} ms")
    print(f"  same hits: {list_hits == hits} ({hits})")

    few = [42, 7, 19, 7, 88, 3]
    print(f"order of {few}:")
    print("  set           ", set(few))
    print("  dict.fromkeys ", list(dict.fromkeys(few)))
    print("  sorted(set)   ", sorted(set(few)))

    for m in (20_000, 40_000, 80_000):
        scans = [rng.randrange(m // 2) for _ in range(m)]   # many repeated scans
        t0 = time.perf_counter()
        a = dedup_with_list(scans)
        t1 = time.perf_counter()
        b = dedup_with_dict(scans)
        t2 = time.perf_counter()
        print(f"dedup {m:6d} scans: list {(t1 - t0) * 1e3:9.2f} ms, "
              f"dict {(t2 - t1) * 1e3:5.2f} ms, equal: {a == b}")


if __name__ == "__main__":
    main()
```

```
contains: 100000 ids, 10000 queries
  list        6462.39 ms
  set            1.30 ms
  sorted         4.93 ms
  same hits: True (4961)
order of [42, 7, 19, 7, 88, 3]:
  set           {3, 7, 42, 19, 88}
  dict.fromkeys  [42, 7, 19, 88, 3]
  sorted(set)    [3, 7, 19, 42, 88]
dedup  20000 scans: list    327.45 ms, dict  0.83 ms, equal: True
dedup  40000 scans: list   1669.68 ms, dict  1.87 ms, equal: True
dedup  80000 scans: list   6025.80 ms, dict  6.00 ms, equal: True
```

The measurements speak plainly: for 10,000 questions, `ArrayList` 1,470 ms against `HashSet` 5.5 ms, a factor of about 270; in Python, 6.5 *seconds* against 1.3 ms, a factor of about 5,000. The sorted structures lie between (`TreeSet` 15.8 ms, `bisect` 4.9 ms). All candidates found the same hits (4,999 in Java, 4,961 in Python, whose random numbers differ from Java's for the same seed).

Removing repeated scans shows $O(n^2)$ against $O(n)$: each doubling of the scans multiplied the `List` time by 3.2 and 3.5 in Java and 5.1 and 3.6 in Python, near the factor 4 of a square law, while `LinkedHashSet` and `dict` stayed within a few milliseconds, with equal results.

Line by line. In **Listing 12.45**, `new Random(42)` fixes the seed and `Collections.shuffle` mixes the numbers so the list cannot profit from a lucky order. `timeContains` takes the test as an `IntPredicate`, supplied by `list::contains`, `hash::contains` and `tree::contains`. The scans are drawn from `m / 2` numbers, so many repeat. In **Listing 12.46**, `as_set.__contains__` is the method behind `in`, and `in_sorted` is `bisect_left` plus one comparison.

The small example deserves a second look. `HashSet` printed `[19, 3, 7, 88, 42]`, bucket order, neither sorted nor as inserted, and the same on every run because a small integer's hash code is

the integer itself, which makes it tempting to trust. Python's `set` printed yet another order. Only insertion-ordered (`LinkedHashSet`, `dict.fromkeys`) and sorted (`TreeSet`, `sorted`) collections *promise* an order.

A tempting wrong solution. The first solution everyone writes is `if (!seen.contains(id)) seen.add(id);` with an `ArrayList`: correct, and instant on twenty scans. But `contains` walks the list, so n scans cost up to $n(n+1)/2$ steps, the sum of Equation (12.2): under a second for 80,000 scans in Java, six seconds in Python, quadrupling with each doubling. The fix changes one type: `LinkedHashSet` keeps the first-seen order *and* answers `contains` in constant time.

> **CHECK YOURSELF.** The counter also needs "the ready orders whose number lies between 150,000 and 150,100". Which of the three sets can answer that without looking at all 100,000 numbers?
>
> *Answer.* Only the sorted one. `TreeSet.subSet(150000, 150101)` finds the start in $O(\log n)$ steps and then walks the matching numbers; in Python, two `bisect_left` calls on the sorted list give the slice boundaries. A `HashSet` has no notion of "between" and would have to test all 100,000 numbers (or all 101 candidates, one `contains` each, which works only because the numbers are integers).

12.8.2 Problem 12.12 — A generic class with a bounded type parameter

> **PROBLEM.** A washing machine's drum is loaded from a bin. Write a generic class `Bin<T>` that holds items of one kind `T` up to a capacity in kilograms: `add(item)` puts the item in only if it still fits and reports whether it did, `load()` returns the current weight, and `heaviest()` returns the heaviest item, typed as `T`, not as `Object`. Every item must have a weight, so `T` must be restricted to types that have one. Also write a method `totalKg` that adds up the weight of a list of *any* kind of weighed thing (a list of shirts, a list of towels), and a generic method `maxOf` that returns the largest element of any list whose elements can be compared.

In our own words. A bin of shirts accepts only shirts, and its heaviest item is a shirt, with its `owner`, without a cast. The bin calls `kg()` on its items, so it must know every `T` has one; the helpers should accept as many kinds of list as is safe.

Design. "Every `T` has a weight" is a *bounded type parameter*, `<T extends Weighed>`, with `Weighed` an interface declaring `double kg()`; inside the class, `item.kg()` is then allowed. The helpers use the other two tools of Chapter 9 [13]:

- `totalKg(List<? extends Weighed> things)` uses a *wildcard*. A `List<Towel>` is **not** a `List<Weighed>` (the wrong solution shows why); `List<? extends Weighed>`, "a list of some type that is `Weighed`", accepts both, at the price that we may only read from it.
- `<T extends Comparable<? super T>> T maxOf(List<? extends T> xs)` is a *generic method*, whose `T` is inferred at each call (`Integer`, `String`, …); the bound says a `T` can be compared with a `T` or with a supertype of it.

The items are *records* (Java 16 and later): `record Shirt(String owner, double kg) implements Weighed { }` declares a final class with private final fields, a constructor, the accessors `owner()` and `kg()`, and `equals`, `hashCode` and `toString`, all generated; `kg()` is exactly what the interface requires.

The Java solution. Listing 12.47.

Listing 12.47 `GenericBin.java`: a bounded generic class `Bin<T extends Weighed>`, a wildcard method `totalKg` and a generic method `maxOf`, with checks.

```java
// GenericBin.java - Problem 12.12: a generic class with a bounded type parameter,
// a wildcard method and a generic method. A washing-machine bin holds items.
import java.util.ArrayList;
import java.util.List;

public class GenericBin {

    /** Anything that has a weight. */
    interface Weighed {
        double kg();
    }

    record Shirt(String owner, double kg) implements Weighed { }

    record Towel(double kg) implements Weighed { }

    /** A bin of items of ONE kind T; T must be Weighed, so kg() may be called. */
    static final class Bin<T extends Weighed> {
        private final double capacityKg;
        private final List<T> items = new ArrayList<>();

        Bin(double capacityKg) {
            this.capacityKg = capacityKg;
        }

        /** Adds the item if it still fits; returns whether it was added. */
        boolean add(T item) {
            if (load() + item.kg() > capacityKg) {
                return false;
            }
            items.add(item);
            return true;
        }

        double load() {
            return totalKg(items);                    // a List<T> is a List<? extends
Weighed>
        }
```

Listing 12.47 (Continued) *GenericBin.java: a bounded generic class Bin<T extends Weighed>, a wildcard method totalKg and a generic method maxOf, with checks.*

```
        /** The heaviest item, or null for an empty bin. */
        T heaviest() {
            T best = null;
            for (T item : items) {
                if (best == null || item.kg() > best.kg()) {
                    best = item;
                }
            }
            return best;
        }

        List<T> items() {
            return List.copyOf(items);              // a read-only copy
        }
    }

    /** Total weight of a list of ANY kind of Weighed things. */
    static double totalKg(List<? extends Weighed> things) {
        double sum = 0;
        for (Weighed w : things) {
            sum += w.kg();
        }
        return sum;
    }

    /** The largest element of any list whose elements can be compared. */
    static <T extends Comparable<? super T>> T maxOf(List<? extends T> xs) {
        T best = xs.get(0);
        for (T x : xs) {
            if (x.compareTo(best) > 0) {
                best = x;
            }
        }
        return best;
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }
```

Listing 12.47 (Continued) *GenericBin.java: a bounded generic class Bin<T extends Weighed>, a wildcard method totalKg and a generic method maxOf, with checks.*

```
    public static void main(String[] args) {
        Bin<Shirt> shirts = new Bin<>(1.0);
        check("add 0.4", true, shirts.add(new Shirt("Ani", 0.4)));
        check("add 0.5", true, shirts.add(new Shirt("Budi", 0.5)));
        check("0.3 does not fit", false, shirts.add(new Shirt("Citra", 0.3)));
        check("heaviest", "Budi", shirts.heaviest().owner());   // no cast: heaviest() is a
Shirt
        check("empty bin", null, new Bin<Towel>(5.0).heaviest());

        List<Towel> towels = List.of(new Towel(0.8), new Towel(1.2));
        check("totalKg of shirts", 0.9, totalKg(shirts.items()));
        check("totalKg of towels", 2.0, totalKg(towels));
        check("maxOf ints", 9, maxOf(List.of(3, 9, 4)));
        check("maxOf strings", "wash", maxOf(List.of("iron", "dry", "wash")));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
PASS add 0.4
PASS add 0.5
PASS 0.3 does not fit
PASS heaviest
PASS empty bin
PASS totalKg of shirts
PASS totalKg of towels
PASS maxOf ints
PASS maxOf strings
All 9 checks passed.
```

The Python solution. Python's generics are *type hints*, checked by a tool such as mypy [14] and ignored when the program runs. **Listing 12.48** states the same design: `Weighed` is a `Protocol` (any class with `kg()` qualifies), `T = TypeVar("T", bound=Weighed)` is the bounded parameter, `Bin(Generic[T])` the generic class, and `total_kg` takes a `Sequence[Weighed]`. Since nothing is enforced at run time, `add` checks `isinstance(item, Weighed)` itself, which `@runtime_checkable` allows.

Listing 12.48 `generic_bin.py`: the same design with `Protocol`, `TypeVar` and `Generic`, checked by mypy before the run and by `isinstance` during it.

```
# generic_bin.py - Problem 12.12: a generic class with a bounded type variable,
# a function over any Weighed things, and a generic function.
# The type hints are checked by a type checker such as mypy, not by Python itself.
import unittest
from dataclasses import dataclass
from typing import Generic, Optional, Protocol, Sequence, TypeVar, runtime_checkable


@runtime_checkable
class Weighed(Protocol):
    """Anything that has a weight."""

    def kg(self) -> float: ...


@dataclass(frozen=True)
class Shirt:
    owner: str
    weight: float

    def kg(self) -> float:
        return self.weight


@dataclass(frozen=True)
class Towel:
    weight: float

    def kg(self) -> float:
        return self.weight


T = TypeVar("T", bound=Weighed)          # T may be any type that is Weighed


class Bin(Generic[T]):
    """A bin of items of ONE kind T."""

    def __init__(self, capacity_kg: float) -> None:
        self._capacity_kg = capacity_kg
        self._items: list[T] = []
```

Listing 12.48 (Continued) *generic_bin.py: the same design with Protocol, TypeVar and Generic, checked by mypy before the run and by isinstance during it.*

```
    def add(self, item: T) -> bool:
        """Add the item if it still fits; return whether it was added."""
        if not isinstance(item, Weighed):    # Python checks nothing unless we ask
            raise TypeError(f"not Weighed: {item!r}")
        if self.load() + item.kg() > self._capacity_kg:
            return False
        self._items.append(item)
        return True

    def load(self) -> float:
        return total_kg(self._items)

    def heaviest(self) -> Optional[T]:
        """The heaviest item, or None for an empty bin."""
        return max(self._items, key=lambda item: item.kg(), default=None)

    def items(self) -> tuple[T, ...]:
        return tuple(self._items)            # a read-only copy


def total_kg(things: Sequence[Weighed]) -> float:
    """Total weight of a sequence of ANY kind of Weighed things."""
    return sum(w.kg() for w in things)


C = TypeVar("C", int, str)               # the element types this function accepts


def max_of(xs: Sequence[C]) -> C:
    """The largest element of a non-empty sequence."""
    best = xs[0]
    for x in xs:
        if x > best:
            best = x
    return best
```

Listing 12.48 (Continued) *generic_bin.py: the same design with Protocol, TypeVar and Generic, checked by mypy before the run and by isinstance during it.*

```
class TestGenericBin(unittest.TestCase):
    def test_bin(self) -> None:
        shirts: Bin[Shirt] = Bin(1.0)
        self.assertTrue(shirts.add(Shirt("Ani", 0.4)))
        self.assertTrue(shirts.add(Shirt("Budi", 0.5)))
        self.assertFalse(shirts.add(Shirt("Citra", 0.3)))
        heaviest = shirts.heaviest()
        self.assertEqual("Budi", heaviest.owner if heaviest else None)
        self.assertIsNone(Bin[Towel](5.0).heaviest())

    def test_wrong_item_is_refused_at_run_time(self) -> None:
        with self.assertRaises(TypeError):
            Bin[Towel](5.0).add("socks")  # type: ignore[arg-type]  # wrong on purpose

    def test_functions(self) -> None:
        shirts = [Shirt("Ani", 0.4), Shirt("Budi", 0.5)]
        self.assertEqual(0.9, total_kg(shirts))
        self.assertEqual(2.0, total_kg([Towel(0.8), Towel(1.2)]))
        self.assertEqual(9, max_of([3, 9, 4]))
        self.assertEqual("wash", max_of(["iron", "dry", "wash"]))


if __name__ == "__main__":
    unittest.main(verbosity=2)
```

```
test_bin (__main__.TestGenericBin.test_bin) ... ok
test_functions (__main__.TestGenericBin.test_functions) ... ok
test_wrong_item_is_refused_at_run_time
(__main__.TestGenericBin.test_wrong_item_is_refused_at_run_time) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Running the type checker on the file, `python -m mypy generic_bin.py`, reports:

```
Success: no issues found in 1 source file
```

The test that adds a string to a bin of towels does so *on purpose*, to check the run-time guard; its comment `# type: ignore[arg-type]` tells mypy the error is intended.

Line by line. In **Listing 12.47**, `add` refuses an item that would exceed the capacity *before* changing anything. `load()` passes the `List<T>` to `totalKg`, allowed because `T extends Weighed`. `heaviest()` returns `T`, so the test writes `shirts.heaviest().owner()` without a cast, and `items()` returns an unmodifiable `List.copyOf`. `maxOf` expects a non-empty list, like `Collections.max`. The test may compare `0.4 + 0.5` with `0.9` exactly because both give the same `double`; in general, compare floating-point sums with a tolerance. In **Listing 12.48**, `@dataclass(frozen=True)` plays the record, `max(..., default=None)` returns `None` for an empty bin, and `C = TypeVar("C", int, str)` is a *constrained* type variable.

A tempting wrong solution. The natural first signature of the helper is `static double totalKg(List<Weighed> things)`. It compiles, and it works when called with a `List<Weighed>`. Called with a `List<Towel>`, javac stops:

```
WrongTotal.java:12: error: incompatible types: List<Towel> cannot be converted to
List<Weighed>
        System.out.println(totalKg(towels));
                                   ^
```

Why? A `List<Weighed>` allows `add(new Shirt(...))`; if a `List<Towel>` could be passed as one, the method could put a shirt into the caller's list of towels. Java generics are therefore *invariant*, and `? extends Weighed` restores flexibility by giving up writing. mypy applies the same rule to Python's `list` and suggests the fix:

```
wrong_total.py:9: error: Argument 1 to "total_kg" has incompatible type "list[Towel]";
expected "list[Weighed]"  [arg-type]
wrong_total.py:9: note: "list" is invariant -- see
https://mypy.readthedocs.io/en/stable/common_issues.html#variance
wrong_total.py:9: note: Consider using "Sequence" instead, which is covariant
Found 1 error in 1 file (checked 1 source file)
```

`Sequence` is read-only, so it may accept a sequence of any subtype, the reasoning behind `? extends` [14]. Python itself ran the wrong version and printed the right total; only the type checker sees the problem.

> **CHECK YOURSELF.** Write the signature of a Java method `addTowels` that adds three new towels to a list given by the caller, and that accepts a `List<Towel>`, a `List<Weighed>` and a `List<Object>`.
>
> *Answer.* `static void addTowels(List<? super Towel> into)`. A list of `Towel` *or of any supertype of it* can safely receive towels. This is the other half of the rule known as "producer `extends`, consumer `super`" [7]: read from `? extends`, write into `? super`.

12.8.3 Problem 12.13 — A chain of comparators

> **PROBLEM.** The pickup board of Profitina Laundry lists the day's orders. Express orders come first; among orders of the same kind, the one that is ready earlier comes first; among those, the customer's name decides alphabetically; and if all that is equal, the order number. Sort six sample orders this way in Java with a `Comparator` chain and in Python with a sort key. Also sort the orders by weight.

In our own words. Four keys, consulted in turn: a later key matters only when all earlier ones are equal. Only the first goes "backwards" (express, `true`, first).

Design. Java builds the order from pieces [5]: `Comparator.comparing(Order::express, Comparator.reverseOrder())` puts `true` first, then `.thenComparingInt(Order::readyAt)`, `.thenComparing(Order::customer)` and `.thenComparingInt(Order::id)`, each stating its own direction. Python uses a key returning a tuple, `(not o.express, o.ready_at, o.customer, o.id)`; tuples compare element by element, and `not o.express` is `False`, which sorts first, for express orders. **Table 12.10** works the sample by hand.

Table 12.10 The six sample orders in board order, with the sort key of each. Orders 105 and 102 are tied on the first two keys and separated by the customer's name.

Id	Customer	Express	Ready	Python key	Position
104	Citra	yes	11	`(False, 11, 'Citra', 104)`	1
105	Ani	yes	17	`(False, 17, 'Ani', 105)`	2
102	Budi	yes	17	`(False, 17, 'Budi', 102)`	3
106	Budi	no	9	`(True, 9, 'Budi', 106)`	4
103	Ani	no	15	`(True, 15, 'Ani', 103)`	5
101	Dewi	no	15	`(True, 15, 'Dewi', 101)`	6

The Java solution. **Listing 12.49** defines the order as a `record` and the board order as a constant `Comparator`, together with two wrong comparators.

Listing 12.49 `OrderSorting.java`: the pickup board sorted by a four-key `Comparator` chain, with two tempting wrong comparators.

```
// OrderSorting.java - Problem 12.13: a Comparator chain for the pickup board.
// Express orders first; then the earlier ready time; then the customer; then the id.
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class OrderSorting {

    record Order(int id, String customer, boolean express, int readyAt, double kg) { }

    /** Right: each key says its own direction; only the express key is reversed. */
    static final Comparator<Order> BOARD =
            Comparator.comparing(Order::express, Comparator.reverseOrder())
                    .thenComparingInt(Order::readyAt)
                    .thenComparing(Order::customer)
                    .thenComparingInt(Order::id);

    /** Tempting but wrong: reversed() at the end reverses the WHOLE chain. */
    static final Comparator<Order> WRONG =
            Comparator.comparing(Order::express)
                    .thenComparingInt(Order::readyAt)
                    .thenComparing(Order::customer)
                    .thenComparingInt(Order::id)
                    .reversed();

    /** Tempting but wrong: subtracting weights and casting drops the fraction. */
    static final Comparator<Order> BY_KG_WRONG = (a, b) -> (int) (a.kg() - b.kg());

    static List<Integer> ids(List<Order> orders) {
        List<Integer> result = new ArrayList<>();
        for (Order o : orders) {
            result.add(o.id());
        }
        return result;
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }
```

Listing 12.49 (Continued) *OrderSorting.java: the pickup board sorted by a four-key Comparator chain, with two tempting wrong comparators.*

```
    public static void main(String[] args) {
        List<Order> orders = List.of(
                new Order(101, "Dewi", false, 15, 4.5),
                new Order(102, "Budi", true, 17, 2.1),
                new Order(103, "Ani", false, 15, 6.0),
                new Order(104, "Citra", true, 11, 2.4),
                new Order(105, "Ani", true, 17, 3.0),
                new Order(106, "Budi", false, 9, 5.2));

        List<Order> board = new ArrayList<>(orders);
        board.sort(BOARD);
        for (Order o : board) {
            System.out.printf("%d %-5s %-7s ready %2d:00%n",
                    o.id(), o.customer(), o.express() ? "express" : "normal", o.readyAt());
        }
        check("board order", List.of(104, 105, 102, 106, 103, 101), ids(board));

        List<Order> wrong = new ArrayList<>(orders);
        wrong.sort(WRONG);
        System.out.println("reversed() at the end: " + ids(wrong));

        List<Order> byKg = new ArrayList<>(orders);
        byKg.sort(BY_KG_WRONG);
        System.out.println("(int) subtraction    : " + ids(byKg));
        byKg.sort(Comparator.comparingDouble(Order::kg));
        check("by kg, right", List.of(102, 104, 105, 101, 106, 103), ids(byKg));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
104 Citra express ready 11:00
105 Ani   express ready 17:00
102 Budi  express ready 17:00
106 Budi  normal  ready  9:00
103 Ani   normal  ready 15:00
101 Dewi  normal  ready 15:00
PASS board order
reversed() at the end: [102, 105, 104, 101, 103, 106]
(int) subtraction    : [102, 104, 105, 101, 103, 106]
PASS by kg, right
All 2 checks passed.
```

The Python solution. **Listing 12.50** uses the tuple key, and a second technique: *stable* sorts from the least to the most important key. Python's `sorted` is guaranteed stable [4], keeping the existing order of equal elements, so the minor order survives inside each group.

Listing 12.50 `order_sorting.py`: the same board with a tuple key and with two stable sorts, and the same two wrong orders.

```
# order_sorting.py - Problem 12.13: sort keys for the pickup board.
# Express orders first; then the earlier ready time; then the customer; then the id.
import sys
import unittest
from dataclasses import dataclass
from functools import cmp_to_key


@dataclass(frozen=True)
class Order:
    id: int
    customer: str
    express: bool
    ready_at: int
    kg: float


ORDERS = [Order(101, "Dewi", False, 15, 4.5), Order(102, "Budi", True, 17, 2.1),
          Order(103, "Ani", False, 15, 6.0), Order(104, "Citra", True, 11, 2.4),
          Order(105, "Ani", True, 17, 3.0), Order(106, "Budi", False, 9, 5.2)]


def board_key(o):
    """Right: a tuple of keys, compared left to right; 'not express' puts express first."""
    return (not o.express, o.ready_at, o.customer, o.id)


def board_two_passes(orders):
    """Also right: stable sorts, from the LAST key to the first."""
    result = sorted(orders, key=lambda o: (o.ready_at, o.customer, o.id))
    return sorted(result, key=lambda o: o.express, reverse=True)


def wrong_board(orders):
    """Tempting but wrong: reverse=True reverses EVERY key of the tuple."""
    return sorted(orders, key=lambda o: (o.express, o.ready_at, o.customer, o.id),
reverse=True)


def by_kg_wrong(a, b):
    """Tempting but wrong: subtracting weights and truncating drops the fraction."""
    return int(a.kg - b.kg)


def ids(orders):
    return [o.id for o in orders]
```

Listing 12.50 (Continued) *order_sorting.py: the same board with a tuple key and with two stable sorts, and the same two wrong orders.*

```
class TestBoard(unittest.TestCase):
    def test_board(self):
        self.assertEqual([104, 105, 102, 106, 103, 101], ids(sorted(ORDERS,
key=board_key)))

    def test_two_passes_agree(self):
        self.assertEqual(ids(sorted(ORDERS, key=board_key)), ids(board_two_passes(ORDERS)))

    def test_by_kg(self):
        self.assertEqual([102, 104, 105, 101, 106, 103], ids(sorted(ORDERS, key=lambda o:
o.kg)))


if __name__ == "__main__":
    for o in sorted(ORDERS, key=board_key):
        kind = "express" if o.express else "normal"
        print(f"{o.id} {o.customer:<5} {kind:<7} ready {o.ready_at:2d}:00")
    print("reverse=True         :", ids(wrong_board(ORDERS)))
    print("int() subtraction    :", ids(sorted(ORDERS, key=cmp_to_key(by_kg_wrong))))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
104 Citra express ready 11:00
105 Ani   express ready 17:00
102 Budi  express ready 17:00
106 Budi  normal  ready  9:00
103 Ani   normal  ready 15:00
101 Dewi  normal  ready 15:00
reverse=True         : [102, 105, 104, 101, 103, 106]
int() subtraction    : [102, 104, 105, 101, 103, 106]
test_board (__main__.TestBoard.test_board) ... ok
test_by_kg (__main__.TestBoard.test_by_kg) ... ok
test_two_passes_agree (__main__.TestBoard.test_two_passes_agree) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Line by line. The record `Order` generates the accessors that the method references use. `board.sort(BOARD)` sorts a copy, because `List.of(...)` is unmodifiable; `%-5s` and `%2d` align the columns. `Comparator.comparingDouble(Order::kg)` is the correct weight order. In **Listing 12.50**, `board_two_passes` sorts by the minor keys and then by `express` in reverse, and `cmp_to_key` turns the old-style wrong comparison into a key.

A tempting wrong solution. The tempting chain writes every key forwards and adds `.reversed()` at the end "to get express first". The result, `[102, 105, 104, 101, 103, 106]`, has express first but *later* ready times first and names from Z to A: `.reversed()` reversed the whole chain. `reverse=True` does the same to a tuple key. Reverse only the key that needs it.

The second, `(a, b) -> (int) (a.kg() - b.kg())`, truncates: 6.0 minus 5.2 is 0.8, cast to 0, "equal", so 103 (6.0 kg) stays before 106 (5.2 kg). With integers, subtraction can overflow instead. `comparingDouble` and `Integer.compare` have neither flaw.

> **CHECK YOURSELF.** Write the Python key that sorts the orders by weight, heaviest first, and among equal weights by id in increasing order.
>
> *Answer.* `key=lambda o: (-o.kg, o.id)`. The minus sign reverses only the weight. `reverse=True` would reverse the ids too. In Java:

> `Comparator.comparingDouble(Order::kg).reversed().thenComparingInt(Order::id);` here `.reversed()` is correct because it is applied *before* the second key is added.

12.9 New Worked Problems: Events, Observers, and MVC

The keypad of Problem 12.10 hides its lessons among many details. The last three problems isolate three of them: why state lives in a model, how an observer goes wrong, and why buttons made in a loop need care.

12.9.1 Problem 12.14 — An event-driven counter with a tested model

> **PROBLEM.** The front desk of Profitina Laundry counts the laundry bags handed in today. Write a small window with three buttons, `+1 bag`, `-1 bag` and `reset`, and two labels: `Bags today: N` and `Machines needed: M`, where one washing machine takes 8 bags. The count may never go below zero. Organize the program as model, views and controller, test the model without any window, and give the program a `--selftest` mode that clicks the buttons from code.

In our own words. One number, three ways to change it, two ways to show it: the number and its rules belong to a model, each label is a view, the buttons are the controller.

Design. Machines are bags divided by 8, *rounded up*: 9 bags need 2. With integer division, which rounds down,

$$\lceil \frac{b}{8} \rceil = \lfloor \frac{b+7}{8} \rfloor \tag{12.10}$$

for $b \geq 0$ bags: $b = 8$ gives $\lfloor 15/8 \rfloor = 1$, $b = 9$ gives 2, $b = 0$ gives 0. Equation (12.10) is `(bags + BAGS_PER_MACHINE - 1) / BAGS_PER_MACHINE` in Java; Python's `-(-bags // 8)` rounds the negated value down to the same effect.

The model notifies its listeners only when the count really changes (`remove()` on an empty counter notifies nobody), and both views read the *same* model, so they cannot disagree.

The Java solution. Listing 12.51 keeps the model, the two views, the controller and a wrong panel in one file.

Listing 12.51 `BagCounterApp.java`: a counter in MVC form with two views, a self-test that exercises the model alone and then clicks the real buttons, and a panel without a model for comparison.

```
// BagCounterApp.java - Problem 12.14: an event-driven counter in MVC form.
// The front desk of Profitina Laundry counts today's bags; two views show the
// count and the number of washing machines needed. Run with --selftest to test.
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.util.ArrayList;
import java.util.List;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class BagCounterApp {

    /** The model: today's number of bags. It knows nothing about Swing. */
    static final class BagCounter {
        static final int BAGS_PER_MACHINE = 8;
        private int bags = 0;
        private final List<Runnable> listeners = new ArrayList<>();

        void addListener(Runnable listener) {
            listeners.add(listener);
        }

        int bags() {
            return bags;
        }

        int machinesNeeded() {
            return (bags + BAGS_PER_MACHINE - 1) / BAGS_PER_MACHINE;   // rounded up
        }

        void add() {
            bags++;
            changed();
        }

        void remove() {
            if (bags > 0) {                          // never below zero
                bags--;
                changed();
            }
        }
```

Listing 12.51 (Continued) *BagCounterApp.java: a counter in MVC form with two views, a self-test that exercises the model alone and then clicks the real buttons, and a panel without a model for comparison.*

```
        void reset() {
            bags = 0;
            changed();
        }

        private void changed() {
            for (Runnable listener : listeners) {
                listener.run();
            }
        }
    }

    /** View 1: the count. */
    static final class CountView extends JLabel {
        private final BagCounter model;

        CountView(BagCounter model) {
            this.model = model;
            update();
        }

        void update() {
            setText("Bags today: " + model.bags());
        }
    }

    /** View 2: the machines. It shares the model, not the other view. */
    static final class MachineView extends JLabel {
        private final BagCounter model;

        MachineView(BagCounter model) {
            this.model = model;
            update();
        }

        void update() {
            setText("Machines needed: " + model.machinesNeeded());
        }
    }

    /** The controller: three buttons that only call the model. */
    static final class CounterButtons extends JPanel {
        final JButton plus = new JButton("+1 bag");
        final JButton minus = new JButton("-1 bag");
        final JButton reset = new JButton("reset");
```

Listing 12.51 (Continued) *BagCounterApp.java: a counter in MVC form with two views, a self-test that exercises the model alone and then clicks the real buttons, and a panel without a model for comparison.*

```
        CounterButtons(BagCounter model) {
            super(new GridLayout(1, 3));
            plus.addActionListener(e -> model.add());
            minus.addActionListener(e -> model.remove());
            reset.addActionListener(e -> model.reset());
            add(plus);
            add(minus);
            add(reset);
        }
    }

    /** Tempting but wrong: the count lives in the label's text; no model. */
    static final class WrongCounterPanel extends JPanel {
        final JLabel count = new JLabel("Bags today: 0");
        final JLabel machines = new JLabel("Machines needed: 0");
        final JButton plus = new JButton("+1 bag");

        WrongCounterPanel() {
            plus.addActionListener(e -> {
                int n = Integer.parseInt(count.getText().substring("Bags today:
".length()));
                count.setText("Bags today: " + (n + 1));  // and the machines label?
forgotten
            });
            add(count);
            add(machines);
            add(plus);
        }
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }
```

Listing 12.51 (Continued) *BagCounterApp.java: a counter in MVC form with two views, a self-test that exercises the model alone and then clicks the real buttons, and a panel without a model for comparison.*

```
    static void selfTest() {
        BagCounter m = new BagCounter();              // the model alone, no Swing at all
        int[] calls = {0};
        m.addListener(() -> calls[0]++);
        m.remove();
        check("not below zero", 0, m.bags());
        check("no change, no update", 0, calls[0]);
        for (int i = 0; i < 9; i++) {
            m.add();
        }
        check("nine bags", 9, m.bags());
        check("two machines", 2, m.machinesNeeded());
        m.reset();
        check("reset", 0, m.machinesNeeded());

        BagCounter model = new BagCounter();          // now model + views + controller
        CountView countView = new CountView(model);
        MachineView machineView = new MachineView(model);
        model.addListener(countView::update);
        model.addListener(machineView::update);
        CounterButtons buttons = new CounterButtons(model);
        for (int i = 0; i < 9; i++) {
            buttons.plus.doClick();
        }
        buttons.minus.doClick();
        System.out.println("MVC  : " + countView.getText() + " | " +
machineView.getText());
        check("views agree", "Machines needed: 1", machineView.getText());

        WrongCounterPanel wrong = new WrongCounterPanel();
        for (int i = 0; i < 9; i++) {
            wrong.plus.doClick();
        }
        System.out.println("wrong: " + wrong.count.getText() + " | " +
wrong.machines.getText());
        System.out.println("All " + passed + " checks passed.");
    }
```

Listing 12.51 (Continued) *BagCounterApp.java: a counter in MVC form with two views, a self-test that exercises the model alone and then clicks the real buttons, and a panel without a model for comparison.*

```
    public static void main(String[] args) {
        if (args.length > 0 && args[0].equals("--selftest")) {
            selfTest();
            return;
        }
        SwingUtilities.invokeLater(() -> {
            BagCounter model = new BagCounter();
            CountView countView = new CountView(model);
            MachineView machineView = new MachineView(model);
            model.addListener(countView::update);
            model.addListener(machineView::update);
            JFrame frame = new JFrame("Bag Counter");
            JPanel views = new JPanel(new GridLayout(2, 1));
            views.add(countView);
            views.add(machineView);
            frame.add(views, BorderLayout.CENTER);
            frame.add(new CounterButtons(model), BorderLayout.SOUTH);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.pack();
            frame.setVisible(true);
        });
    }
}
```

`java BagCounterApp --selftest` prints:

```
PASS not below zero
PASS no change, no update
PASS nine bags
PASS two machines
PASS reset
MVC  : Bags today: 8 | Machines needed: 1
PASS views agree
wrong: Bags today: 9 | Machines needed: 0
All 6 checks passed.
```

The Python solution. In **Listing 12.52**, a view turns the model into text with one function and hands it to another, `show`, which writes on a Tkinter label in the window and stores the text in a dictionary in the tests; model and views are tested without Tkinter.

Listing 12.52 `bag_counter.py`: the same counter in Python; a view receives the function that displays its text, so tests can replace the label.

```
# bag_counter.py - Problem 12.14: an event-driven counter in MVC form.
# The front desk of Profitina Laundry counts today's bags; two views show the
# count and the number of washing machines needed. Run with --selftest to test.
import sys
import unittest


class BagCounter:
    """The model: today's number of bags. It knows nothing about Tkinter."""
    BAGS_PER_MACHINE = 8

    def __init__(self):
        self._bags = 0
        self._listeners = []

    def add_listener(self, listener):
        self._listeners.append(listener)

    def bags(self):
        return self._bags

    def machines_needed(self):
        return -(-self._bags // self.BAGS_PER_MACHINE)     # rounded up

    def add(self):
        self._bags += 1
        self._changed()

    def remove(self):
        if self._bags > 0:                                 # never below zero
            self._bags -= 1
            self._changed()

    def reset(self):
        self._bags = 0
        self._changed()

    def _changed(self):
        for listener in self._listeners:
            listener()


class LabelView:
    """A view: it turns the model into text and hands the text to 'show'.
    'show' puts it on a Tkinter label, or keeps it for a test."""
```

Listing 12.52 (Continued) *bag_counter.py: the same counter in Python; a view receives the function that displays its text, so tests can replace the label.*

```
    def __init__(self, model, make_text, show):
        self._model, self._make_text, self._show = model, make_text, show
        self.update()

    def update(self):
        self._show(self._make_text(self._model))


def count_text(model):
    return f"Bags today: {model.bags()}"


def machine_text(model):
    return f"Machines needed: {model.machines_needed()}"


def wrong_plus(labels):
    """Tempting but wrong: the count lives in the label's text; no model."""
    n = int(labels["count"][len("Bags today: "):])
    labels["count"] = f"Bags today: {n + 1}"              # and the machines label?
forgotten


class TestBagCounter(unittest.TestCase):
    def test_model_alone(self):
        m = BagCounter()
        calls = []
        m.add_listener(lambda: calls.append(1))
        m.remove()
        self.assertEqual(0, m.bags())
        self.assertEqual([], calls)                       # no change, no update
        for _ in range(9):
            m.add()
        self.assertEqual((9, 2), (m.bags(), m.machines_needed()))
        m.reset()
        self.assertEqual(0, m.machines_needed())
```

Listing 12.52 (Continued) *bag_counter.py: the same counter in Python; a view receives the function that displays its text, so tests can replace the label.*

```
    def test_views_agree(self):
        model, shown = BagCounter(), {}
        views = [LabelView(model, count_text, lambda t: shown.__setitem__("count", t)),
                 LabelView(model, machine_text, lambda t: shown.__setitem__("machines",
t))]
        for v in views:
            model.add_listener(v.update)
        for _ in range(9):
            model.add()                                  # what the "+1 bag" button calls
        model.remove()
        self.assertEqual({"count": "Bags today: 8", "machines": "Machines needed: 1"},
shown)


def run_window():
    import tkinter as tk
    root = tk.Tk()
    root.title("Bag Counter")
    model = BagCounter()
    for make_text in (count_text, machine_text):
        label = tk.Label(root, width=24)
        label.pack()
        model.add_listener(LabelView(model, make_text, lambda t, lb=label:
lb.config(text=t)).update)
    buttons = tk.Frame(root)
    buttons.pack()
    for text, action in (("+1 bag", model.add), ("-1 bag", model.remove), ("reset",
model.reset)):
        tk.Button(buttons, text=text, command=action).pack(side="left")
    root.mainloop()


if __name__ == "__main__":
    if sys.argv[1:] == ["--selftest"]:
        labels = {"count": "Bags today: 0", "machines": "Machines needed: 0"}
        for _ in range(9):
            wrong_plus(labels)
        print("wrong:", labels["count"], "|", labels["machines"])
        sys.stdout.flush()
        unittest.main(argv=[sys.argv[0]], verbosity=2)
    else:
        run_window()
```

`python bag_counter.py --selftest` prints:

```
wrong: Bags today: 9 | Machines needed: 0
test_model_alone (__main__.TestBagCounter.test_model_alone) ... ok
test_views_agree (__main__.TestBagCounter.test_views_agree) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Line by line. In **Listing 12.51**, `BagCounter` imports nothing from Swing; a web page could use it unchanged. `add`, `remove` and `reset` end with `changed()`, `remove` only inside its `if`. Each view sets its label in `update` and calls it once in its constructor, so the label is right before the first click; the controller only connects buttons to model methods. The self-test first checks the model's rules alone, including "no change, no update", then assembles the real Swing objects, clicks nine times `+1 bag` and once `-1 bag`, and checks that both views show 8 bags and 1 machine.

In **Listing 12.52**, `LabelView.update()` is simply `show(make_text(model))`. In `run_window`, `lambda t, lb=label: lb.config(text=t)` binds the *current* label as a default argument (see Problem 12.16), and `unittest.main(argv=[sys.argv[0]], ...)` keeps unittest from reading `--selftest` as its own option.

A tempting wrong solution. The quickest program has no model: the `+1 bag` listener parses the number out of the label, adds one and writes it back. `WrongCounterPanel` and `wrong_plus` do this, and after nine clicks show `Bags today: 9 | Machines needed: 0`: nothing reminded the listener of the second label, whereas a model's `changed()` updates every view. The count also lives in text formatted for people, so renaming the label to `Bags: 9` breaks the parsing, and nothing can be tested without clicking.

> **CHECK YOURSELF.** The owner wants a third view, a label that turns red when more than 40 bags are waiting. Which classes change?
>
> *Answer.* None of the existing ones. Add a new view class (a `JLabel` whose `update` sets the text and the foreground color from `model.bags()`), create it, and register its `update` with `model.addListener`. The model and the controller are untouched, which is the point of MVC.

12.9.2 Problem 12.15 — Two observer bugs

> **PROBLEM.** A detergent tank of Profitina Laundry notifies its listeners whenever detergent is used. Three listeners are registered: a one-shot alarm that rings once when the level falls below 10 liters and then unsubscribes itself, a display that shows the level, and a logger. The tank starts at 14 liters and three washes use 3 liters each. The first version of the tank has two bugs: the display ends up showing a level the tank does not have, and the notifications misbehave when the alarm unsubscribes. Find both bugs, explain them, and fix them.

In our own words. After three washes the tank holds $14 - 3 \cdot 3 = 5$ liters, the display shows 5, the alarm has rung once (at 8), and the logger has seen 11, 8 and 5.

Design. Notification has two rules that are easy to break [1]:

1. **Change the state first, then notify.** Listeners are called to look at the *new* state; notifying first leaves every view one step behind.
2. **Do not let the listener list change under the loop that walks it.** A listener may unsubscribe itself (a one-shot alarm) or subscribe others while notified, so the loop must walk a *copy* taken before the first call.

`Tank` contains both versions, selected by the flag `careful`, so one run shows bugs and fixes side by side.

The Java solution. Listing 12.53.

Listing 12.53 `ObserverBugs.java`: a subject that notifies before changing and walks its live list of listeners, next to the fixed version.

```
// ObserverBugs.java - Problem 12.15: two observer bugs and their fixes.
// A detergent tank reports its level; a display and a one-shot alarm listen.
import java.util.ArrayList;
import java.util.ConcurrentModificationException;
import java.util.List;
import java.util.function.IntConsumer;

public class ObserverBugs {

    /** The subject. 'careful' switches between the buggy and the fixed code. */
    static final class Tank {
        private int liters;
        private final boolean careful;
        private final List<IntConsumer> listeners = new ArrayList<>();

        Tank(int liters, boolean careful) {
            this.liters = liters;
            this.careful = careful;
        }

        void addListener(IntConsumer l) {
            listeners.add(l);
        }

        void removeListener(IntConsumer l) {
            listeners.remove(l);
        }

        int liters() {
            return liters;
        }

        void use(int amount) {
            if (careful) {
                liters -= amount;                          // FIX 1: change the state first,
                for (IntConsumer l : List.copyOf(listeners)) {   // FIX 2: then notify a
copy
                    l.accept(liters);
                }
            } else {
                for (IntConsumer l : listeners) {          // BUG 2: the live list
                    l.accept(liters);                      // BUG 1: before the change
                }
                liters -= amount;
            }
        }
    }
```

Listing 12.53 (Continued) *ObserverBugs.java: a subject that notifies before changing and walks its live list of listeners, next to the fixed version.*

```
    /** A one-shot alarm: below 10 liters it rings once and unsubscribes itself. */
    static final class LowAlarm implements IntConsumer {
        private final Tank tank;
        private final List<String> log;

        LowAlarm(Tank tank, List<String> log) {
            this.tank = tank;
            this.log = log;
        }

        @Override
        public void accept(int level) {
            if (tank.liters() < 10) {
                log.add("ALARM at " + tank.liters() + " l");
                tank.removeListener(this);              // unsubscribes DURING the
notification
            }
        }
    }

    static List<String> run(boolean careful) {
        Tank tank = new Tank(14, careful);
        List<String> log = new ArrayList<>();
        int[] shown = {tank.liters()};                  // what the display shows
        tank.addListener(new LowAlarm(tank, log));
        tank.addListener(level -> shown[0] = level);
        tank.addListener(level -> log.add("logger saw " + level));
        try {
            for (int i = 0; i < 3; i++) {
                tank.use(3);
            }
        } catch (ConcurrentModificationException e) {
            log.add("ConcurrentModificationException");
        }
        log.add("display shows " + shown[0] + " l, tank has " + tank.liters() + " l");
        return log;
    }

    private static int passed = 0;
```

Listing 12.53 (Continued) *ObserverBugs.java: a subject that notifies before changing and walks its live list of listeners, next to the fixed version.*

```
    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        System.out.println("buggy tank:");
        List<String> buggy = run(false);
        buggy.forEach(line -> System.out.println("  " + line));
        System.out.println("careful tank:");
        List<String> careful = run(true);
        careful.forEach(line -> System.out.println("  " + line));
        check("buggy: stale display", "display shows 11 l, tank has 8 l",
buggy.get(buggy.size() - 1));
        check("careful: alarm once", 1L, careful.stream().filter(l ->
l.startsWith("ALARM")).count());
        check("careful: display agrees", "display shows 5 l, tank has 5 l",
careful.get(careful.size() - 1));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
buggy tank:
  logger saw 14
  logger saw 11
  ALARM at 8 l
  ConcurrentModificationException
  display shows 11 l, tank has 8 l
careful tank:
  logger saw 11
  ALARM at 8 l
  logger saw 8
  logger saw 5
  display shows 5 l, tank has 5 l
PASS buggy: stale display
PASS careful: alarm once
PASS careful: display agrees
All 3 checks passed.
```

The Python solution. Listing 12.54.

Listing 12.54 `observer_bugs.py`: the same subject in Python, where the second bug fails silently.

```
# observer_bugs.py - Problem 12.15: two observer bugs and their fixes.
# A detergent tank reports its level; a display and a one-shot alarm listen.
import sys
import unittest


class Tank:
    """The subject. 'careful' switches between the buggy and the fixed code."""

    def __init__(self, liters, careful):
        self._liters = liters
        self._careful = careful
        self._listeners = []

    def add_listener(self, listener):
        self._listeners.append(listener)

    def remove_listener(self, listener):
        self._listeners.remove(listener)

    def liters(self):
        return self._liters

    def use(self, amount):
        if self._careful:
            self._liters -= amount                    # FIX 1: change the state first,
            for listener in list(self._listeners):    # FIX 2: then notify a copy
                listener(self._liters)
        else:
            for listener in self._listeners:          # BUG 2: the live list
                listener(self._liters)                # BUG 1: before the change
            self._liters -= amount


class LowAlarm:
    """A one-shot alarm: below 10 liters it rings once and unsubscribes itself."""

    def __init__(self, tank, log):
        self._tank, self._log = tank, log

    def __call__(self, level):
        if self._tank.liters() < 10:
            self._log.append(f"ALARM at {self._tank.liters()} l")
            self._tank.remove_listener(self)          # unsubscribes DURING the
notification
```

Listing 12.54 (Continued) *observer_bugs.py: the same subject in Python, where the second bug fails silently.*

```
def run(careful):
    tank = Tank(14, careful)
    log = []
    shown = [tank.liters()]                          # what the display shows

    def display(level):
        shown[0] = level

    tank.add_listener(LowAlarm(tank, log))
    tank.add_listener(display)
    tank.add_listener(lambda level: log.append(f"logger saw {level}"))
    for _ in range(3):
        tank.use(3)
    log.append(f"display shows {shown[0]} l, tank has {tank.liters()} l")
    return log


class TestTank(unittest.TestCase):
    def test_buggy_display_is_stale(self):
        self.assertEqual("display shows 11 l, tank has 5 l", run(False)[-1])

    def test_careful(self):
        log = run(True)
        self.assertEqual(1, sum(1 for line in log if line.startswith("ALARM")))
        self.assertEqual("display shows 5 l, tank has 5 l", log[-1])


if __name__ == "__main__":
    print("buggy tank:")
    for line in run(False):
        print("  " + line)
    print("careful tank:")
    for line in run(True):
        print("  " + line)
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
buggy tank:
  logger saw 14
  logger saw 11
  ALARM at 8 l
  logger saw 8
  display shows 11 l, tank has 5 l
careful tank:
  logger saw 11
  ALARM at 8 l
  logger saw 8
  logger saw 5
  display shows 5 l, tank has 5 l
test_buggy_display_is_stale (__main__.TestTank.test_buggy_display_is_stale) ... ok
test_careful (__main__.TestTank.test_careful) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Line by line. The buggy branch notifies first, so the logger records 14 and 11, levels from before the washes. On the third wash the old level is 8, and the alarm rings and removes itself while the loop walks the list. Java's `ArrayList` detects the change and the loop's next step throws `ConcurrentModificationException`: the display misses that wash, and since the exception leaves `use` before the subtraction, the level is not even reduced: `display shows 11 l, tank has 8 l`. Python's `for` simply continues at the next index; the alarm was at index 0, the display moved

there, and the loop, now at index 1, *skips it.* No error, a silently stale display: `display shows 11 l, tank has 5 l`. The quiet version of the bug is the more dangerous one. The careful branch subtracts first and walks `List.copyOf(listeners)` or `list(self._listeners)`, so every listener sees the new level and the alarm can leave the real list safely. `LowAlarm` reads `tank.liters()`, not its argument, as real listeners often do, which is exactly why "change first" matters.

A tempting wrong solution. The tempting fix for the exception is to catch it, or to loop with an index, `for (int i = 0; i < listeners.size(); i++)`. The exception disappears and the program behaves like the Python version, skipping a listener without a word: the message was never the problem, only its honest report. Java's `CopyOnWriteArrayList`, which copies its array on every change so that iterations walk a stable snapshot, is made for listener lists, which are read far more often than changed [5].

> **CHECK YOURSELF.** In the careful tank, a listener *adds* another listener while being notified. Is the new listener called during the same notification?
>
> *Answer.* No. The loop walks the copy taken before the first call, and the new listener is not in it. It will be called from the next notification on. That is usually what one wants: a listener registered during an event should see the *next* event, not a half-finished current one.

12.9.3 Problem 12.16 — Buttons made in a loop

> **PROBLEM.** A keypad has twelve buttons labeled by the characters of `"123456789*0#"`, and each button must record its own character when it is clicked. Create the buttons and their listeners in one loop, in Java and in Python. Show three correct ways to give each listener its own key, and explain the version in which clicking 4, 6, 6, 3 records `####`.

In our own words. A listener runs *later*, when its button is clicked, and must then know its key; by then the loop variable holds something else. Each listener must take its key *with it*.

Design. A lambda or inner class that uses a variable of the surrounding method *captures* it; code plus captured variables form a *closure*. The languages capture differently:

- **Java captures values.** A lambda or anonymous class may use a local variable only if it is *effectively final* [15], never reassigned, and keeps a copy of its value; the compiler rejects the classic bug.
- **Python captures variables.** A `lambda` looks up the variable's value when it *runs*; after a loop, every lambda made in it sees the last value, a famous pitfall [16].

The correct patterns: in Java, a fresh variable per turn (declared inside the loop, or a for-each variable), captured by a lambda, an anonymous class or a named class with a field; in Python, anything that evaluates the key *when the function is made*: a default argument, a factory function, or `functools.partial`.

The Java solution. Listing 12.55 builds four keypads: three correct ones and one with the bug.

Listing 12.55 `KeypadClosures.java`: listeners made in a loop with a lambda, an anonymous inner class and a named class, and a version whose listeners share one mutable cell.

```java
// KeypadClosures.java - Problem 12.16: twelve buttons made in a loop. Each
// listener must remember ITS OWN key. Three listeners that work, one that does not.
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.List;

import javax.swing.JButton;

public class KeypadClosures {

    static final String KEYS = "123456789*0#";

    /** Clicks the buttons of the given keys and returns what the listeners recorded. */
    static String click(List<JButton> buttons, String keys, StringBuilder pressed) {
        pressed.setLength(0);
        for (char k : keys.toCharArray()) {
            buttons.get(KEYS.indexOf(k)).doClick();
        }
        return pressed.toString();
    }

    /** Right 1: a lambda that captures a new, effectively final variable per turn. */
    static List<JButton> withLambdas(StringBuilder pressed) {
        List<JButton> buttons = new ArrayList<>();
        for (int i = 0; i < KEYS.length(); i++) {
            char key = KEYS.charAt(i);                    // a fresh 'key' in every turn
            JButton b = new JButton(String.valueOf(key));
            b.addActionListener(e -> pressed.append(key));
            buttons.add(b);
        }
        return buttons;
    }

    /** Right 2: an anonymous inner class; it, too, copies the captured value. */
    static List<JButton> withInnerClasses(StringBuilder pressed) {
        List<JButton> buttons = new ArrayList<>();
        for (char key : KEYS.toCharArray()) {             // for-each: a fresh 'key' too
            JButton b = new JButton(String.valueOf(key));
            b.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    pressed.append(key);
                }
            });
            buttons.add(b);
```

Listing 12.55 (Continued) *KeypadClosures.java: listeners made in a loop with a lambda, an anonymous inner class and a named class, and a version whose listeners share one mutable cell.*

```
        }
        return buttons;
    }

    /** Right 3: a named listener class that stores its key in a field. */
    static final class KeyListener implements ActionListener {
        private final char key;
        private final StringBuilder pressed;

        KeyListener(char key, StringBuilder pressed) {
            this.key = key;
            this.pressed = pressed;
        }

        @Override
        public void actionPerformed(ActionEvent e) {
            pressed.append(key);
        }
    }

    static List<JButton> withNamedClass(StringBuilder pressed) {
        List<JButton> buttons = new ArrayList<>();
        for (char key : KEYS.toCharArray()) {
            JButton b = new JButton(String.valueOf(key));
            b.addActionListener(new KeyListener(key, pressed));
            buttons.add(b);
        }
        return buttons;
    }

    /** Tempting but wrong: all listeners share ONE mutable cell, read at click time. */
    static List<JButton> withSharedCell(StringBuilder pressed) {
        List<JButton> buttons = new ArrayList<>();
        char[] current = new char[1];                    // the array is effectively
final...
        for (char key : KEYS.toCharArray()) {
            current[0] = key;                            // ...but its content is not
            JButton b = new JButton(String.valueOf(key));
            b.addActionListener(e -> pressed.append(current[0]));
            buttons.add(b);
        }
        return buttons;
    }

    private static int passed = 0;
```

Listing 12.55 (Continued) *KeypadClosures.java: listeners made in a loop with a lambda, an anonymous inner class and a named class, and a version whose listeners share one mutable cell.*

```
    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        StringBuilder pressed = new StringBuilder();
        check("lambdas", "4663", click(withLambdas(pressed), "4663", pressed));
        check("inner classes", "4663", click(withInnerClasses(pressed), "4663", pressed));
        check("named class", "4663", click(withNamedClass(pressed), "4663", pressed));
        System.out.println("shared cell: " + click(withSharedCell(pressed), "4663",
pressed));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
PASS lambdas
PASS inner classes
PASS named class
shared cell: ####
All 3 checks passed.
```

The loop with an index cannot capture `i` at all. Writing `b.addActionListener(e -> pressed.append(keys.charAt(i)));` inside `for (int i = 0; ...; i++)` is rejected by the compiler:

```
LoopIndex.java:8: error: local variables referenced from a lambda expression must be final
or effectively final
            b.addActionListener(e -> pressed.append(keys.charAt(i)));
                                                                  ^
1 error
```

because `i++` reassigns `i`: Java turns Python's run-time surprise into a compile-time error.

The Python solution. Listing 12.56.

Listing 12.56 `keypad_closures.py`: commands made in a loop, one version with the late-binding bug and three correct ones.

```
# keypad_closures.py - Problem 12.16: twelve button commands made in a loop.
# Each command must remember ITS OWN key. One tempting version that does not.
import sys
import unittest
from functools import partial

KEYS = "123456789*0#"
pressed = []                                    # what the buttons have recorded


def click(commands, keys):
    """Call the command of each key, as clicking its button would; return what was
pressed."""
    pressed.clear()
    for k in keys:
        commands[KEYS.index(k)]()
    return "".join(pressed)


def wrong_commands():
    """Tempting but wrong: every lambda looks up 'key' when it RUNS, not when it is
made."""
    return [lambda: pressed.append(key) for key in KEYS]


def with_default_argument():
    """Right 1: a default value is evaluated when the lambda is MADE."""
    return [lambda key=key: pressed.append(key) for key in KEYS]


def make_command(key):
    return lambda: pressed.append(key)          # every call has its own 'key'


def with_factory():
    """Right 2: a factory function (the make_command of Problem 12.10)."""
    return [make_command(key) for key in KEYS]


def with_partial():
    """Right 3: functools.partial stores the argument now and calls later."""
    return [partial(pressed.append, key) for key in KEYS]
```

Listing 12.56 (Continued) *keypad_closures.py: commands made in a loop, one version with the late-binding bug and three correct ones.*

```
class TestClosures(unittest.TestCase):
    def test_right_versions(self):
        for make in (with_default_argument, with_factory, with_partial):
            self.assertEqual("4663", click(make(), "4663"))

    def test_wrong_version(self):
        self.assertEqual("####", click(wrong_commands(), "4663"))


if __name__ == "__main__":
    print("wrong commands :", click(wrong_commands(), "4663"))
    print("default arg    :", click(with_default_argument(), "4663"))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
wrong commands : ####
default arg    : 4663
test_right_versions (__main__.TestClosures.test_right_versions) ... ok
test_wrong_version (__main__.TestClosures.test_wrong_version) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Line by line. In **Listing 12.55**, `withLambdas` declares `char key` inside the loop body, twelve variables each assigned once; `withInnerClasses` uses a for-each variable, fresh in every turn, captured by an anonymous `ActionListener` under the same rule (the only way to write listeners before Java 8); `withNamedClass` makes the capture explicit in a `private final` field, which is what the compiler does behind the scenes for the other two. In **Listing 12.56**, the lambdas of `wrong_commands` all refer to the comprehension's variable `key`, which is `"#"` by the time they run. `lambda key=key:` evaluates the default when the lambda is made; `make_command(key)` works because each *call* creates a new local `key` (the helper of Problem 12.10); `partial(pressed.append, key)` stores the argument for later.

A tempting wrong solution. Java programmers sometimes "fix" the effectively-final error with a one-element array: the reference is final, its content is not. `withSharedCell` writes each key into `current[0]` and every listener reads `current[0]` *when clicked*, by which time it holds `#`: clicking 4, 6, 6, 3 records `####`, Python's bug reintroduced by hand. The compiler's rule protected the program from shared mutable state; give each listener its *own* copy.

> **CHECK YOURSELF.** In Python, `buttons = [tk.Button(root, text=k, command=lambda: press(k)) for k in "0123"]`. Which key does each button press, and what is the smallest change that fixes it?
>
> *Answer.* All four buttons press `"3"`, the last value of `k`. The smallest fix is a default argument: `command=lambda k=k: press(k)`.

12.10 The Problems of This Chapter Compared

Table 12.11 lays the sixteen problems side by side. As in Chapter 8, read down the last column: every tempting wrong solution passes the obvious test and fails on one particular input or one particular sequence of events.

Table 12.11 The sixteen problems of the chapter: what each practices and the case its tempting wrong solution gets wrong.

Problem	Topic	Main idea	What the wrong solution misses
12.1	strings	one key per letter; `StringBuffer` grows in place	four-letter keys (s, v, y, z)
12.2	files, sets	a set removes duplicates; the file is closed always	capitals and spaces in the file
12.3	command line	`args` versus `sys.argv`; sorted output	arrays print as `[L...@hash`
12.4	sorted list	numerical order without conversion; binary search plus a walk	several words per signature; 17-digit signatures
12.5	interface	a contract from the users' needs	leaking the first implementation's list
12.6	timing	build once, then look up many times	measuring start-up instead of lookups
12.7	hash map	find-or-create the drawer; read-only results	`put` overwrites; callers clear the drawer
12.8	trie	store each word along its path; trim results	words stored only at the end: no prefixes
12.9	interface use	one line chooses the implementation	variables typed with the concrete class
12.10	MVC	state in the model, recomputed not duplicated	text area as data; signature stored twice
12.11	collections	choose by operation; measure	`List.contains` in a loop: $O(n^2)$
12.12	generics	bounded type parameter; `? extends`	`List<Towel>` is not a `List<Weighed>`
12.13	comparators	each key states its own direction	`.reversed()` at the end; `(int)` subtraction
12.14	MVC	two views of one model; tested without a window	a second view forgotten
12.15	observer	change first, notify a copy	stale views; skipped listeners
12.16	closures	each listener takes its own key	late binding: every button presses `#`

12.11 Profitina in Practice

Profitina, the AI-visibility (GEO) platform in production at app.profitina.com, measures how ready a brand's website is to be cited by AI answer engines and helps fix what it finds. Its public features show the ideas of this chapter at work in a real product.

The right collection for the frequent question. A crawler meets the same page under many spellings of its address (a trailing slash, tracking parameters, letter case). "Have I seen this page?" is Problem 12.11's membership question, asked for every link on every page, so it belongs in a hash set of canonical addresses, never in a list. The weekly score trend per brand is the opposite case, a series read in time order and exported to CSV, the job of an ordered structure.

One interface, many implementations. The scanner examines a site through *signals*, such as how extractable its content is and whether it publishes structured data, each scored and explained on its own behind one common interface: the shape of `Dictionary` and its three implementations. A new signal is a new class, not an edit of the loop.

Models, views and observers. The dashboard gathers notifications, followed channels, rank and activity, and each channel shows an "N new" counter of top-level posts: views of a model, like the labels of Problem 12.14. The real-time community feed pushes changes to every open page with

Server-Sent Events, the observer pattern of Problem 12.15 across a network, and *optimistic posting* follows its first rule: the composer clears and the post appears at once because the local model changes first; if sending fails, the model is restored and the text returns to the composer, so no view shows something that did not happen.

The system suggests; the person chooses. Predictive text offers candidates and the user confirms one. Profitina's AI features follow the same rule: drafts and scores are advice with an explanation, never a guarantee. The runnable version of every idea here is this chapter's Profitina Laundry code.

12.12 Common Mistakes, and How to Avoid Them

Choosing a collection by habit. A list is the first collection everyone learns, and it is the wrong one for "is this present?" asked many times. Write down the frequent operations first (Section 12.2) and match each to a structure (Problems 12.4, 12.7, 12.11).

Comparing numbers as strings, or strings as numbers. `"22" < "3"` as strings; `Double.parseDouble` loses digits beyond sixteen. Signatures are compared by length and then as strings, Equation (12.4) (Problem 12.4).

Keeping only the first binary-search hit. A search stops at *some* match; collect its neighbors (Problem 12.4).

Handing out a collection that belongs to an object. A caller who receives the map's own set can change the map. Return a read-only view (`Collections.unmodifiableSet`, `List.copyOf`) or a copy (`frozenset`, `tuple`) (Problems 12.7, 12.10).

Replacing instead of adding in a multi-valued map. `put(key, newSet)` for every value keeps only the last one. Use `computeIfAbsent(...).add(...)` in Java and `setdefault(...).add(...)` in Python (Problem 12.7).

Printing a set and expecting an order. Hash sets have no meaningful order, and Python even changes the order of string sets between runs. Sort before printing, or use an ordered collection (Problems 12.3, 12.11).

Using a raw or concrete type where the interface would do. `DictionaryListImpl d` ties code to one implementation; `Dictionary d` does not. `List<Weighed>` refuses a `List<Towel>`; `List<? extends Weighed>` accepts it (Problems 12.9, 12.12).

Reversing a whole comparator chain, or subtracting to compare. Put `reverseOrder()` on the one key that needs it; use `Integer.compare` and `comparingDouble` rather than subtraction (Problem 12.13).

Keeping state in the view. A label or a text area is for showing, not for storing. Put the state in a model with ordinary methods and test it without a window (Problems 12.10, 12.14).

Notifying before changing, or walking a live list of listeners. Change the state first, then notify a copy of the listener list (Problem 12.15).

Capturing a loop variable that keeps changing. In Python, bind the value with a default argument, a factory function or `partial`; in Java, give each listener its own final variable or field, and never share a mutable cell (Problem 12.16).

Timing the wrong thing. One tiny measurement times the clock, and `time` on a small input times the program's start-up. Measure the phase that differs, with large data, several times (Problem 12.6).

12.13 Chapter Summary

- A problem about collections is a problem about operations: list them with their frequencies, match each frequent one to a structure, hide the choice behind an interface, keep the state in a model, test it without a window, and confirm with measurements (Section 12.2, **Table 12.1**).
- The signature of a word, Equation (12.1), is computed with a table or a switch and built in a `StringBuffer` or a list joined once, because repeated string concatenation copies $n(n+1)/2$ characters, Equation (12.2) (Problem 12.1).
- The prototype reads the file for every question, Equation (12.3); it is the yardstick of correctness and speed (Problems 12.2, 12.3).
- A sorted list of `WordSig` pairs, ordered by Equation (12.4), answers with a binary search of at most Equation (12.5) looks plus a walk to the neighbors, Equation (12.6); the `Dictionary` interface contains only what its users need (Problems 12.4, 12.5).
- Timing must separate building from looking up; for many lookups on a large dictionary, the prototype is tens of thousands of times slower than the list (Problem 12.6, **Table 12.5**).
- A `HashMap` from signatures to sets of words looks up and inserts in expected constant time, Equation (12.8); a trie stores each word along its path, Equation (12.9), and is the only one of the four that answers prefixes (Problems 12.7, 12.8, **Table 12.7**, **Figure 12.4**).
- A GUI in MVC keeps its state in a model that stores each fact once; views only show, controllers only forward; `--selftest` modes and fake dictionaries let everything except the pixels be tested (Problem 12.10).
- Collections are chosen by operation and confirmed by measurement (12.11); generic classes use bounded type parameters and wildcards, because generic types are invariant (12.12); comparator chains state each key's direction (12.13); two views of one model cannot disagree, Equation (12.10) (12.14); observers need "change first, notify a copy" (12.15); listeners made in a loop must capture their own values (12.16).

12.14 Exercises

Exercises 12.1 to 12.5 ask for code; their solutions are programs in Section 12.15. The others ask for reasoning and are answered in words.

1. **(Equality in a hash set.)** Put two objects `new WordSig("4663", "home")` into a `HashSet`. How many elements does the set have, and why? Write a class `Pair` with `equals` and `hashCode` (Python: `__eq__` and `__hash__`) so that the set has one element.
2. **(A table instead of a switch.)** Write `tableSignature(String word)` that uses a `Map<Character, Character>` built once, in a static initializer, instead of `keyOf`. Test that it agrees with `wordToSignature` for all 26 letters, for `"Don't"` and for `"3rd"`.
3. **(Prefixes from a sorted map.)** Store the dictionary in a `TreeMap<String, Set<String>>` keyed by signature in plain string order, and write `signatureToWords` so that it returns every *whole* word whose signature *starts with* the given one, using `subMap`. What does it return for 9675? Why does the upper bound of the range end with the character `:`?

4. **(Inverting a map.)** Write a generic method `<K, V> Map<V, Set<K>> invert(Map<K, V> m)` that turns "key to value" into "value to set of keys". Apply it to a map from five words to their signatures, and look up 5477.
5. **(Longest first.)** Write a `Comparator<String>` (Python: a key function) that orders words by length, longest first, and words of equal length alphabetically. Sort `go, home, a, wash, laundry, is`.
6. **(Counting looks.)** For a dictionary of 400,000 words, at most how many cards does the binary search of Problem 12.4 examine? How many nodes does the tree of Problem 12.8 visit to find the node of 96753? How many drawers does the map of Problem 12.7 open?
7. **(Memory of the tree.)** How many word references does `DictionaryTreeImpl` store for `words.txt`, and for the 200,000-word invented dictionary? Use Equation (12.9); the numbers you need are in Section 12.6.2.
8. **(Fewer updates.)** `DictionaryModel.press` notifies the listeners after every press, even when nothing changed. Describe how to notify only when the text on show changes. In **Listing 12.39**, the model receives 16 presses; how many notifications would the improved model send?
9. **(A punctuation key.)** The owner of Profitina Laundry wants key 1 to type a full stop after the current word. Which of the four GUI classes change, and how? Does the rule "no spaces or signatures in the model" allow it?
10. **(Your own race.)** Generate a 400,000-word dictionary with `MakeDictionary 400000 big400.txt` and run `TimingComparison` on it. Before running, predict for each of the four rows whether the time per lookup should roughly stay the same, grow a little, or double, compared with the 200,000-word run. Then compare with your measurement and with **Figure 12.4**.

12.15 Solutions to the Exercises

Exercises 1 to 5. **Listing 12.57** and **Listing 12.58** contain all the requested code, checked against the answers below. The Java file belongs to the package `predictive`, because it uses `WordSig` and `PredictivePrototype`; run it with `java predictive.ExerciseSolutions` from the `java` folder.

Listing 12.57 `ExerciseSolutions.java`: a pair with `equals` and `hashCode`, a table-driven signature, a `TreeMap` dictionary with prefix ranges, a generic `invert`, and a comparator, with checks.

```java
// ExerciseSolutions.java - solutions to the programming exercises of Chapter 12.
package predictive;

import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.Set;
import java.util.TreeMap;
import java.util.TreeSet;

public class ExerciseSolutions {

    /** Exercise 12.1: a pair with equals and hashCode, so a HashSet sees duplicates. */
    static final class Pair {
        private final String signature;
        private final String word;

        Pair(String signature, String word) {
            this.signature = signature;
            this.word = word;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) {
                return true;
            }
            if (!(o instanceof Pair other)) {
                return false;
            }
            return signature.equals(other.signature) && word.equals(other.word);
        }

        @Override
        public int hashCode() {
            return Objects.hash(signature, word);    // equal pairs, equal hash codes
        }
    }
```

Listing 12.57 (Continued) *ExerciseSolutions.java: a pair with equals and hashCode, a table-driven signature, a TreeMap dictionary with prefix ranges, a generic invert, and a comparator, with checks.*

```
    /** Exercise 12.2: a table built once, used for every letter. */
    private static final Map<Character, Character> KEY_OF = new HashMap<>();

    static {
        String[] letters = {"abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
        for (int k = 0; k < letters.length; k++) {
            for (char c : letters[k].toCharArray()) {
                KEY_OF.put(c, (char) ('2' + k));
            }
        }
    }

    static String tableSignature(String word) {
        StringBuilder sb = new StringBuilder();
        for (char c : word.toLowerCase().toCharArray()) {
            sb.append(KEY_OF.getOrDefault(c, ' '));
        }
        return sb.toString();
    }

    /** Exercise 12.3: a sorted map answers "every signature that starts with s". */
    static final class SortedMapDictionary implements Dictionary {
        private final TreeMap<String, Set<String>> map = new TreeMap<>();

        SortedMapDictionary(List<String> words) {
            for (String w : words) {
                map.computeIfAbsent(PredictivePrototype.wordToSignature(w), k -> new
HashSet<>()).add(w);
            }
        }

        /** Whole words whose signature starts with the given one; ':' follows '9'. */
        @Override
        public Set<String> signatureToWords(String signature) {
            Set<String> result = new HashSet<>();
            for (Set<String> words : map.subMap(signature, signature + ":").values()) {
                result.addAll(words);
            }
            return result;
        }
    }
```

Listing 12.57 (Continued) *ExerciseSolutions.java: a pair with equals and hashCode, a table-driven signature, a TreeMap dictionary with prefix ranges, a generic invert, and a comparator, with checks.*

```
    /** Exercise 12.4: turn key -> value into value -> set of keys. */
    static <K, V> Map<V, Set<K>> invert(Map<K, V> m) {
        Map<V, Set<K>> result = new HashMap<>();
        for (Map.Entry<K, V> e : m.entrySet()) {
            result.computeIfAbsent(e.getValue(), v -> new HashSet<>()).add(e.getKey());
        }
        return result;
    }

    /** Exercise 12.5: longer words first, equal lengths alphabetically. */
    static final Comparator<String> LONGEST_FIRST =

Comparator.comparingInt(String::length).reversed().thenComparing(Comparator.naturalOrder())
;

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) throws Exception {
        Set<WordSig> plain = new HashSet<>(List.of(new WordSig("4663", "home"), new
WordSig("4663", "home")));
        Set<Pair> pairs = new HashSet<>(List.of(new Pair("4663", "home"), new Pair("4663",
"home")));
        check("12.1 WordSig duplicates kept", 2, plain.size());
        check("12.1 Pair duplicates removed", 1, pairs.size());

        boolean same = true;
        for (char c = 'a'; c <= 'z'; c++) {
            String s = String.valueOf(c);
            same &= tableSignature(s).equals(PredictivePrototype.wordToSignature(s));
        }
        check("12.2 all letters agree", true, same);
        check("12.2 other characters", "366 8", tableSignature("Don't"));
        check("12.2 digits", " 73", tableSignature("3rd"));
```

Listing 12.57 (Continued) *ExerciseSolutions.java: a pair with equals and hashCode, a table-driven signature, a TreeMap dictionary with prefix ranges, a generic invert, and a comparator, with checks.*

```
        List<String> words = new ArrayList<>();
        for (String line : Files.readAllLines(Path.of("words.txt"))) {
            String w = line.trim().toLowerCase();
            if (PredictivePrototype.isValidWord(w)) {
                words.add(w);
            }
        }
        Dictionary sorted = new SortedMapDictionary(words);
        check("12.3 prefix 9675", "[work, worked, worker, works, world, worlds]",
                new TreeSet<>(sorted.signatureToWords("9675")).toString());
        check("12.3 whole words: goods", true,
sorted.signatureToWords("4663").contains("goods"));

        Map<String, String> wordToSig = new HashMap<>();
        for (String w : List.of("gone", "home", "kiss", "lips", "cat")) {
            wordToSig.put(w, PredictivePrototype.wordToSignature(w));
        }
        Map<String, Set<String>> sigToWords = invert(wordToSig);
        check("12.4 invert", "[kiss, lips]", new
TreeSet<>(sigToWords.get("5477")).toString());

        List<String> list = new ArrayList<>(List.of("go", "home", "a", "wash", "laundry",
"is"));
        list.sort(LONGEST_FIRST);
        check("12.5 longest first", "[laundry, home, wash, go, is, a]", list.toString());
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
PASS 12.1 WordSig duplicates kept
PASS 12.1 Pair duplicates removed
PASS 12.2 all letters agree
PASS 12.2 other characters
PASS 12.2 digits
PASS 12.3 prefix 9675
PASS 12.3 whole words: goods
PASS 12.4 invert
PASS 12.5 longest first
All 9 checks passed.
```

Listing 12.58 `exercise_solutions.py`: the same solutions in Python; the prefix search uses two `bisect_left` calls on the sorted signatures.

```
# exercise_solutions.py - solutions to the programming exercises of Chapter 12.
import unittest
from bisect import bisect_left

from predictive.dictionary import Dictionary
from predictive.prototype import is_valid_word, word_to_signature
from predictive.word_sig import WordSig


class Pair:
    """Exercise 12.1: a pair with __eq__ and __hash__, so a set sees duplicates."""

    def __init__(self, signature, word):
        self.signature, self.word = signature, word

    def __eq__(self, other):
        if not isinstance(other, Pair):
            return NotImplemented
        return (self.signature, self.word) == (other.signature, other.word)

    def __hash__(self):
        return hash((self.signature, self.word))     # equal pairs, equal hashes


# Exercise 12.2: the table, built once by a dict comprehension.
_KEY = {letter: str(k + 2)
        for k, group in enumerate(["abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv",
"wxyz"])
        for letter in group}


def table_signature(word):
    return "".join(_KEY.get(c, " ") for c in word.lower())


class SortedKeysDictionary(Dictionary):
    """Exercise 12.3: sorted signatures answer "every signature that starts with s"."""

    def __init__(self, words):
        self._map = {}
        for w in words:
            self._map.setdefault(word_to_signature(w), set()).add(w)
        self._keys = sorted(self._map)               # plain string order
```

Listing 12.58 (Continued) *exercise_solutions.py: the same solutions in Python; the prefix search uses two bisect_left calls on the sorted signatures.*

```
    def signature_to_words(self, signature):
        """Whole words whose signature starts with the given one; ':' follows '9'."""
        lo = bisect_left(self._keys, signature)
        hi = bisect_left(self._keys, signature + ":")
        return set().union(*(self._map[k] for k in self._keys[lo:hi]))


def invert(m):
    """Exercise 12.4: turn key -> value into value -> set of keys."""
    result = {}
    for k, v in m.items():
        result.setdefault(v, set()).add(k)
    return result


def longest_first(words):
    """Exercise 12.5: longer words first, equal lengths alphabetically."""
    return sorted(words, key=lambda w: (-len(w), w))


class TestExercises(unittest.TestCase):
    def test_12_1_equality(self):
        self.assertEqual(2, len({WordSig("4663", "home"), WordSig("4663", "home")}))
        self.assertEqual(1, len({Pair("4663", "home"), Pair("4663", "home")}))

    def test_12_2_table(self):
        for c in "abcdefghijklmnopqrstuvwxyz":
            self.assertEqual(word_to_signature(c), table_signature(c))
        self.assertEqual("366 8", table_signature("Don't"))
        self.assertEqual(" 73", table_signature("3rd"))

    def test_12_3_prefixes(self):
        with open("words.txt", encoding="utf-8") as f:
            words = [w for w in (line.strip().lower() for line in f) if is_valid_word(w)]
        d = SortedKeysDictionary(words)
        self.assertEqual(["work", "worked", "worker", "works", "world", "worlds"],
                         sorted(d.signature_to_words("9675")))
        self.assertIn("goods", d.signature_to_words("4663"))

    def test_12_4_invert(self):
        m = {w: word_to_signature(w) for w in ("gone", "home", "kiss", "lips", "cat")}
        self.assertEqual({"kiss", "lips"}, invert(m)["5477"])
```

Listing 12.58 (Continued) *exercise_solutions.py: the same solutions in Python; the prefix search uses two bisect_left calls on the sorted signatures.*

```
    def test_12_5_order(self):
        self.assertEqual(["laundry", "home", "wash", "go", "is", "a"],
                         longest_first(["go", "home", "a", "wash", "laundry", "is"]))


if __name__ == "__main__":
    unittest.main(verbosity=2)
```

```
test_12_1_equality (__main__.TestExercises.test_12_1_equality) ... ok
test_12_2_table (__main__.TestExercises.test_12_2_table) ... ok
test_12_3_prefixes (__main__.TestExercises.test_12_3_prefixes) ... ok
test_12_4_invert (__main__.TestExercises.test_12_4_invert) ... ok
test_12_5_order (__main__.TestExercises.test_12_5_order) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

1. Two elements. `WordSig` inherits `equals` and `hashCode` from `Object`, where objects are equal only if they are the *same* object. `Pair` overrides both, and `Objects.hash(signature, word)` gives equal pairs equal hash codes, the contract `HashSet` relies on [7]; overriding only `equals` would still give two elements, because the objects would land in different buckets. Python's `__eq__` and `__hash__` play the same roles.
2. A *static initializer*, `static { ... }`, runs once when the class is loaded and fills `KEY_OF` from the eight letter groups; `(char) ('2' + k)` is the group's digit, and `getOrDefault(c, ' ')` makes `"3rd"` into `" 73"`. Python builds the table with a two-clause *dict comprehension*.
3. `[work, worked, worker, works, world, worlds]`, uncut. In string order, all strings starting with `9675` lie together, and `:` is the character after `9`, so `"9675:"` is greater than every one of them and smaller than the next signature, such as `9676`. `subMap(s, s + ":")` returns exactly that range, its start found in $O(\log n)$: prefixes from a sorted map, without a hand-made tree.
4. `computeIfAbsent(value, ...)` finds or creates each value's set and adds the key: Problem 12.7's multi-valued map in generic form. Gone and home map to 4663, kiss and lips to 5477, cat to 228, so the lookup of 5477 gives `{kiss, lips}`.
5. `[laundry, home, wash, go, is, a]`. `comparingInt(String::length).reversed()` puts longer words first, and `thenComparing(naturalOrder())`, added *after* `reversed()`, orders equal lengths alphabetically; Python's key `(-len(w), w)` does the same.
6. By Equation (12.5), $\lceil \log_2 400001 \rceil = 19$ cards. The tree takes 5 steps below the root for 96753, whatever the dictionary's size. The map opens exactly one drawer.
7. By Equation (12.9), one reference per letter of every distinct word: 354 references in the 120 non-root nodes for `words.txt`, and 1,183,435 for the 165,929 distinct invented words. The words themselves are stored once; only references repeat.
8. Keep the text before the change (`String before = getText();`) and run the listeners only if `!getText().equals(before)`. Three presses of the test change nothing (the refused `7`, the second `0`, the last `#` on an empty word), so the improved model sends 13 notifications.
9. The controller enables key 1 and labels it; the model's `press('1')` finishes the current word, if any, and appends `"."` to the finished words, which `getText` joins without a space before it. The model still stores no spaces or signatures, since spaces are added only when the text is shown; the view does not change.
10. Doubling the dictionary should roughly double the prototype's time (Equation (12.3)), leave the map's unchanged (Equation (12.8)), add one comparison to the list's 18 (Equation (12.6)), and raise the tree's somewhat, as more words share each node (Equation (12.9)). The runs behind **Figure 12.4** gave, from 200,000 to 400,000 words, a prototype growth of 1.5 times in Java and 2.5 in Python, a map below one microsecond, and a list and tree within their noise. If your map doubles or your prototype stays flat, suspect the experiment first.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

[2] Python Software Foundation, "unittest — Unit testing framework," *The Python Standard Library*, Python 3.14. [Online]. Available: https://docs.python.org/3/library/unittest.html

[3] D. L. Grover, M. T. King, and C. A. Kushler, "Reduced keyboard disambiguating computer," U.S. Patent 5 818 437, Oct. 6, 1998.

[4] Python Software Foundation, "Built-in Types," "bisect — Array bisection algorithm" and "Sorting Techniques," *The Python Standard Library* and *Python HOWTOs*, Python 3.14. [Online]. Available: https://docs.python.org/3/

[5] Oracle, "Java SE 21 API Specification: java.util (Collections, Comparator, HashMap, TreeMap, CopyOnWriteArrayList, Observable)." [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[7] J. Bloch, *Effective Java*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2018.

[8] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[9] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[10] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26–49, 1988.

[11] Python Software Foundation, "tkinter — Python interface to Tcl/Tk," *The Python Standard Library*, Python 3.14. [Online]. Available: https://docs.python.org/3/library/tkinter.html

[12] Oracle, "The Java Tutorials: Creating a GUI With Swing" and "Concurrency in Swing." [Online]. Available: https://docs.oracle.com/javase/tutorial/uiswing/

[13] M. Naftalin and P. Wadler, *Java Generics and Collections*. Sebastopol, CA, USA: O'Reilly Media, 2006.

[14] G. van Rossum, J. Lehtosalo, and Ł. Langa, "PEP 484 — Type Hints," 2014; and The mypy project, "mypy documentation: Generics and Common issues (variance)." [Online]. Available: https://peps.python.org/pep-0484/ and https://mypy.readthedocs.io/

[15] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*. Oracle America, 2023. [Online]. Available: https://docs.oracle.com/javase/specs/

[16] Python Software Foundation, "Programming FAQ: Why do lambdas defined in a loop with different values all return the same result?" [Online]. Available: https://docs.python.org/3/faq/programming.html

The collections, their costs and the advice on `equals`, `hashCode`, read-only views and wildcards follow the Java API documentation [5], Bloch [7] and Naftalin and Wadler [13], with the language rules for generics and effectively final variables from the specification [15]. Binary search and hashing are analyzed in Cormen et al. [6], tries in Knuth [9], and linear congruential generators and the weakness of their low bits in Knuth [8]. The Python material relies on the standard library documentation [4], [2], [11], the programming FAQ [16] and the type-hint specification and mypy [14]. The observer pattern is from Gamma et al. [1], Model–View–Controller from Krasner and Pope [10], Swing's threading rule from the Java tutorial [12], and predictive text from its patent [3].

Chapter 13

Iterators, Immutability, and More Collections

By the end of this chapter you will be able to walk through any collection with an *iterator* in Java and in Python and say exactly what `hasNext()`, `next()`, `__iter__` and `__next__` do at every step; write collections of your own that work in a `for` loop, first by hand and then, in Python, with a *generator*; explain why changing a list while you walk over it throws `ConcurrentModificationException` in Java and quietly skips elements in Python, and change collections safely instead; put objects in any order you like with `Comparable`, `Comparator` and Python's key functions, and spot a comparison that only looks correct; tell an alias from a shallow copy and a shallow copy from a deep one; design *immutable* objects from nothing — `final` fields, unmodifiable collections, records and defensive copies in Java, frozen dataclasses, tuples and `frozenset` in Python — and say why they make programs simpler; explain what a *value object* is and why a key that changes inside a `HashMap` or a `dict` is lost; and replace many loops with *streams* in Java and with comprehensions and `itertools` in Python. Profitina Laundry gets an immutable `Money` type, an order history that you can walk newest-first, and a sales report built as a pipeline.

13.1 Where We Left Off

Chapter 9 introduced the three great families of collections — lists, sets and maps — together with the generics that let a `List<String>` hold only strings. Chapter 10 put windows on the screen, Chapter 11 separated a program into model, view and controller, and Chapter 12 worked through problems that combined all of it. Throughout those chapters we used collections the simple way: we added things to them, looked things up in them, and walked through them with a `for` loop without asking how the loop actually visits each element.

This chapter lifts the lid. The first half is about **walking**: the small object called an *iterator* that every `for` loop over a collection quietly creates, what it can and cannot do, how to write one for a collection of our own, and what goes wrong when a collection changes under an iterator's feet. The middle of the chapter is about **order and copies**: how to sort objects that have no single obvious order, and what it really means to copy a list of objects. The second half is about **immutability**: objects that cannot change after they are made. At first sight an object that cannot change sounds like a handicap. By the end of the chapter it will look like one of the most useful tools we have, because an object that cannot change can be shared, stored as a key, passed to any method and handed to another thread without a moment's worry. We finish with **streams** and comprehensions, which say "keep the big orders and add up their totals" almost as we would say it aloud.

The laundry has grown again. The owner wants a history of every order, a view of one customer's orders, and a weekly report, and prices must never be changed by accident. That is precisely the kind of problem this chapter's tools were made for.

> ✓ **KEY INSIGHT.** An *iterator* separates **walking** a collection from **storing** it; *immutability* separates **creating** an object from **using** it. Both ideas make code simpler for the same reason: each piece of the program has fewer things it needs to know and fewer ways to go wrong.

13.2 Walking Through a Collection: Iterators

Suppose someone hands you a `Collection<String>`. You know it holds strings, and you know how many (`size()`), but you do not know whether it is a list, a set, or something else entirely. How do you print every string in it?

If it were a `List`, you could use an index: `get(0)`, `get(1)`, and so on up to `size() - 1`, because a list keeps its elements in a numbered order. But a `HashSet` has no index; there is no "element number 2" of a set. A tree set, a queue or a collection someone wrote last year may store its elements in ways you cannot see. We need a way of saying "give me the elements one at a time, in whatever order suits you, until there are none left" that works for *every* collection. That way is the iterator.

> **DEFINITION.** An **iterator** is an object that lets us examine each element of a collection in turn, one at a time, without knowing how the collection stores its elements. It remembers its own position, can say whether there are more elements, and can hand over the next one.

The idea is old and was given a name in the famous catalog of design patterns: the *Iterator pattern* "provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation" [1]. The idea goes back to the language CLU of the 1970s, designed by Barbara Liskov's group, whose later Java text still teaches how to specify iterators carefully [2]. Every collection in Java and in Python supplies iterators, and both languages build their `for` loops on top of them.

13.2.1 The Iterator protocol in Java

In Java, an iterator is any object that implements the interface `java.util.Iterator<E>`. Its two essential methods are:

- `boolean hasNext()` — are there more elements to visit?
- `E next()` — return the next element and move past it. If there is none, throw `NoSuchElementException`.

A third method, `void remove()`, is optional; we meet it in Section 13.5. Every collection has a method `iterator()` that creates a fresh iterator positioned before its first element [3]. **Listing 13.1** uses these methods directly.

Listing 13.1 `IteratorBasics.java`: walking a collection with an explicit iterator, changing a local copy of an element, the enhanced `for` loop, and asking for one element too many.

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class IteratorBasics {
    public static void main(String[] args) {
        // 1. Print every string in the collection with an explicit iterator.
        Collection<String> crew = new ArrayList<>();
        crew.add("Luffy, Monkey D.");
        crew.add("Zoro, Roronoa");
        crew.add("Robin, Nico");
        Iterator<String> iter = crew.iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }

        // 2. Changing the local variable does not change the collection.
        System.out.println();
        Collection<String> captain = new ArrayList<>();
        captain.add("Luffy, Monkey D.");
        Iterator<String> it2 = captain.iterator();
        while (it2.hasNext()) {
            String element = it2.next();
            element += " and Zoro, Roronoa";
            System.out.println(element);
        }
        System.out.println("Print captain's content:");
        for (String s : captain) {
            System.out.println(s);
        }

        // 3. The enhanced for loop is an iterator in disguise.
        System.out.println();
        Collection<Integer> c = new ArrayList<>();
        c.add(3);
        c.add(2);
        c.add(4);
        int sum = 0;
        for (Integer i : c) {
            sum += i;
        }
        System.out.println("sum = " + sum);
```

Listing 13.1 (Continued) *IteratorBasics.java: walking a collection with an explicit iterator, changing a local copy of an element, the enhanced for loop, and asking for one element too many.*

```
        // 4. Asking for one element too many.
        Iterator<Integer> it3 = c.iterator();
        it3.next();
        it3.next();
        it3.next();
        System.out.println("hasNext() after three calls to next(): " + it3.hasNext());
        try {
            it3.next();
        } catch (NoSuchElementException e) {
            System.out.println("fourth next(): " + e.getClass().getSimpleName());
        }
    }
}
```

Compile and run it in the chapter's `java` folder with `javac IteratorBasics.java` and `java IteratorBasics`. It prints:

```
Luffy, Monkey D.
Zoro, Roronoa
Robin, Nico

Luffy, Monkey D. and Zoro, Roronoa
Print captain's content:
Luffy, Monkey D.

sum = 9
hasNext() after three calls to next(): false
fourth next(): NoSuchElementException
```

Read the first block slowly, because every iteration in this chapter is a variation of it.

- `Collection<String> crew = new ArrayList<>();` declares the variable with the most general type that does the job, `Collection`, exactly as Chapter 9 recommended. From here on, the code knows only that `crew` is *some* collection.
- `Iterator<String> iter = crew.iterator();` asks the collection for an iterator. The iterator is a separate object; the collection is not changed by creating it, and you may create as many as you like.
- `while (iter.hasNext())` asks, before every step, whether another element exists. With three strings the answer is `true` three times and then `false`.
- `iter.next()` returns the next string *and* moves the iterator on. That second effect is easy to overlook: calling `next()` twice in one loop body would consume two elements.

It helps to picture the iterator's position as a **cursor** that sits *between* elements rather than on them, as **Figure 13.1** shows. A new iterator's cursor sits before the first element. Each call to `next()` jumps the cursor over one element and returns the element it jumped over. When the cursor has passed the last element, `hasNext()` answers `false`.

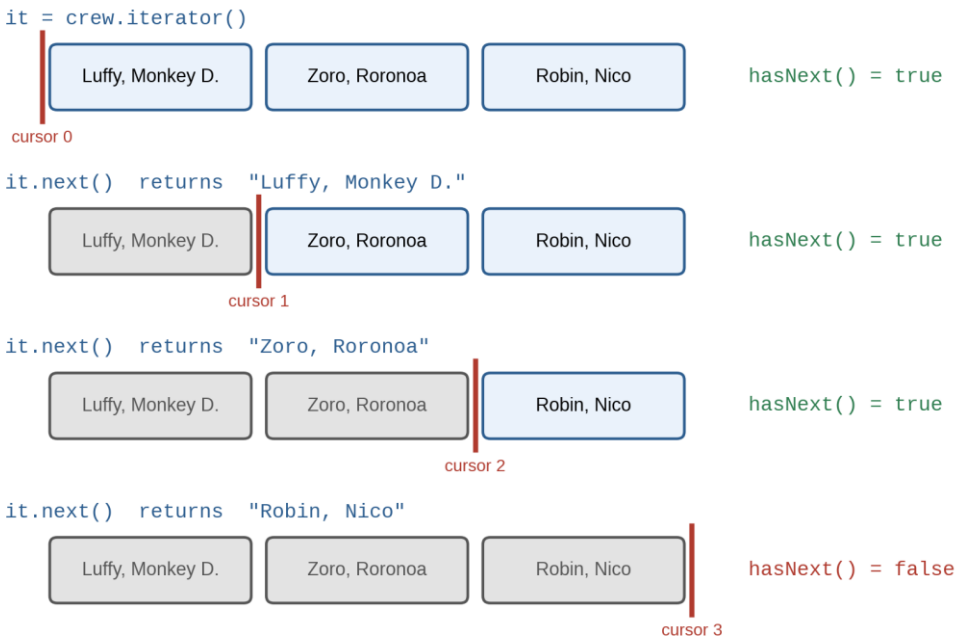


Figure 13.1 An iterator's cursor sits between elements. Each `next()` jumps over one element and returns it (grey boxes are already returned); after the last jump, `hasNext()` is `false`.

If we number the gaps $0, 1, \ldots, n$ for a collection of n elements, the cursor c starts at 0, each `next()` adds one to it, and the whole behavior of `hasNext()` is one comparison:

$$\text{hasNext}() \Leftrightarrow c \neq n \tag{13.1}$$

This tiny formula, Equation (13.1), looks too obvious to write down. Keep it in mind anyway: in Section 13.5 it will explain a bug that silently skips an element.

The second block of **Listing 13.1** shows what an iterator *cannot* do. Inside the loop, `String element = it2.next();` copies a *reference* to the string into the local variable `element`. The statement `element += " and Zoro, Roronoa";` builds a brand-new string and makes `element` refer to it. The collection still holds the old reference, so printing `captain` afterwards shows the unchanged `Luffy, Monkey D.`. Strings are immutable, so there was never any way to change the string itself; and reassigning a local variable never reaches back into the collection. If the elements were *mutable* objects, `element.setName(...)` *would* change the object inside the collection — a distinction that Section 13.7 examines closely.

The third block computes `sum = 9` with an enhanced `for` loop, and the fourth shows the end of the road. After three calls to `next()` on a three-element collection, `hasNext()` is `false`, and a fourth `next()` throws `NoSuchElementException`. A correct program never sees this exception: it asks `hasNext()` first.

13.2.2 The same protocol in Python

Python has exactly the same idea with different names. Any object that can be walked over is an **iterable**; calling the built-in function `iter(x)` on it returns an **iterator**; calling `next(it)` on the iterator returns the next element, and when there are no more elements the iterator raises the exception `StopIteration` [4]. There is no separate `hasNext()`: in Python you simply try to take the next element, and the exception is the answer “there is none”. This design was fixed in 2001 by the proposal that gave Python its iterators, PEP 234 [5]. **Listing 13.2** is the Python twin of **Listing 13.1**.

Listing 13.2 `iterator_basics.py`: `iter()`, `next()` and `StopIteration`, the `for` loop, and asking for one element too many.

```
"""iterator_basics.py: Python's iterator protocol, step by step."""


def main():
    # 1. Print every string in the collection with an explicit iterator.
    crew = ["Luffy, Monkey D.", "Zoro, Roronoa", "Robin, Nico"]
    it = iter(crew)               # ask the iterable for an iterator
    while True:
        try:
            element = next(it)    # the next element, or StopIteration
        except StopIteration:
            break
        print(element)

    # 2. Changing the local variable does not change the collection.
    print()
    captain = ["Luffy, Monkey D."]
    for element in captain:
        element += " and Zoro, Roronoa"
        print(element)
    print("Print captain's content:")
    for s in captain:
        print(s)

    # 3. The for loop does steps 1 and 2 for us.
    print()
    c = [3, 2, 4]
    total = 0
    for i in c:
        total += i
    print("sum =", total)

    # 4. Asking for one element too many.
    it3 = iter(c)
    next(it3)
    next(it3)
    next(it3)
    print("next(it3, 'no more') after three calls:", next(it3, "no more"))
    try:
        next(it3)
    except StopIteration as e:
        print("fourth next():", type(e).__name__)
    print("an iterator is its own iterator:", iter(it3) is it3)


```

Listing 13.2 (Continued) *iterator_basics.py: iter(), next() and StopIteration, the for loop, and asking for one element too many.*

```
if __name__ == "__main__":
    main()
```

Run it with `python3 iterator_basics.py` (on Windows, `python iterator_basics.py` or `py iterator_basics.py`):

```
Luffy, Monkey D.
Zoro, Roronoa
Robin, Nico

Luffy, Monkey D. and Zoro, Roronoa
Print captain's content:
Luffy, Monkey D.

sum = 9
next(it3, 'no more') after three calls: no more
fourth next(): StopIteration
an iterator is its own iterator: True
```

The first block is what a Python `for` loop does behind the scenes. `iter(crew)` asks the list for an iterator; the `while True` loop calls `next(it)` again and again; when `next` raises `StopIteration`, the `except` clause leaves the loop with `break`. Nobody writes loops like this in real code, but it is worth writing once, to see that `for element in crew:` is no magic.

The second and third blocks match Java's: rebinding the loop variable `element` does not change the list, and `for i in c` sums to 9. The fourth block shows two small conveniences. `next(it3, "no more")` takes a *default* that is returned instead of raising `StopIteration`, which is Python's nearest equivalent of checking `hasNext()`. And the last line prints `True` for `iter(it3) is it3`: calling `iter` on an iterator returns the iterator itself. That rule is what lets a Python `for` loop accept either an iterable or an iterator, and we will see in Section 13.4 why it matters.

Table 13.1 puts the two protocols side by side.

Table 13.1 The iteration protocol in Java and Python.

Idea	Java	Python
Something you can walk over	implements `Iterable<E>`	defines `__iter__` (an *iterable*)
Get a fresh iterator	`c.iterator()`	`iter(c)`, which calls `c.__iter__()`
Is there another element?	`it.hasNext()`	no method: try `next` and catch `StopIteration`
Take the next element	`it.next()`	`next(it)`, which calls `it.__next__()`
No more elements	`next()` throws `NoSuchElementException`	`__next__` raises `StopIteration`
Remove the last element returned	`it.remove()` (optional)	not part of the protocol
Loop that uses all of this	`for (E e : c)`	`for e in c:`
Iterator written for you	—	a *generator*: a function containing `yield`

13.2.3 What an iterator cannot do

An iterator is deliberately modest. Through an ordinary `Iterator` you can only move *forward*, one element at a time, and you can only *read* the element you are given. You cannot go back, you cannot jump ahead, and you cannot put a different object into the collection in the place of the one you just received. If you want a collection in which some elements have been replaced, the plain iterator's answer is: make a new collection, and add elements to it as you walk the old one.

That modesty is a feature: because an iterator promises so little, *every* collection can keep the promise — a list, a set, a tree, a file read line by line, even a sequence computed on demand and never stored, as in Section 13.4.

13.3 Iterable, the Enhanced for Loop, and ListIterator

The explicit `while (iter.hasNext())` loop is precise but wordy. Java therefore offers the **enhanced `for` loop** (also called the *for-each loop*), and Python's ordinary `for` loop is exactly the same idea. Any class that implements the interface `java.lang.Iterable<T>` — which has a single method, `Iterator<T> iterator()` — can appear after the colon, and every class that implements `Collection` implements `Iterable` too. The enhanced loop visits the elements in a forward direction and one at a time, as the plain iterator does.

The Java Language Specification defines the enhanced loop by translation [6]. When you write

```
for (Integer i : c) {
    sum += i;
}
```

the compiler turns it into

```
for (Iterator<Integer> $it = c.iterator(); $it.hasNext(); ) {
    Integer i = $it.next();
    sum += i;
}
```

where `$it` stands for a hidden variable you cannot name. So the third block of **Listing 13.1** *is* the first block, written shorter. Python's `for i in c:` is translated the same way into the loop that the first block of **Listing 13.2** wrote out by hand. (For an array, Java's enhanced loop uses an index instead of an iterator, because arrays are not objects that implement `Iterable`.)

The enhanced loop hides the iterator, and that has one consequence you must remember: **inside a for-each loop you have no handle on the iterator**, so you cannot call its `remove()` method. Section 13.5 shows why that matters.

13.3.1 Removing adjacent duplicates

Here is a small task that shows the iterator's limitations and how to live with them. A list is sorted, so equal elements sit next to each other, and we want a new list with each value only once: `[Ace, Luffy, Luffy, Sanji, Sanji, Zoro, Zoro, Zoro]` should become `[Ace, Luffy, Sanji, Zoro]`. With a plain iterator we cannot look backwards, and we cannot delete the element we are looking at from inside a for-each loop. The classic solution is to use a variable to *remember the previous element*, and a *new* list to collect the unique ones. **Listing 13.3** shows that classic version, a safer version, and an in-place version that uses a more capable iterator.

Listing 13.3 `RemoveDuplicates.java`: removing adjacent duplicates from a sorted list with a remembered previous element, a version that handles empty lists and `null`, and an in-place version with `ListIterator`.

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;
import java.util.Objects;

public class RemoveDuplicates {
    // The classic version: remembers the previous element in a variable.
    public static <T> List<T> removeDuplicatesFromSortedList(List<T> input) {
        // Note that this method can't cope with removing duplicate null values
        T previous = null;
        List<T> result = new ArrayList<T>();
        result.add(input.get(0));
        Iterator<T> iter = input.iterator();
        while (iter.hasNext()) {
            T current = iter.next();
            if (previous != null && !previous.equals(current)) {
                result.add(current);
            }
            previous = current;
        }
        return result;
    }

    // A safer version: works for an empty list and for null elements.
    public static <T> List<T> dedupSorted(List<T> input) {
        List<T> result = new ArrayList<>();
        boolean first = true;
        T previous = null;
        for (T current : input) {
            if (first || !Objects.equals(previous, current)) {
                result.add(current);
            }
            previous = current;
            first = false;
        }
        return result;
    }
```

Listing 13.3 (Continued) *RemoveDuplicates.java: removing adjacent duplicates from a sorted list with a remembered previous element, a version that handles empty lists and null, and an in-place version with ListIterator.*

```
    // In place, with a ListIterator: no second list is needed.
    public static <T> void dedupSortedInPlace(List<T> list) {
        ListIterator<T> it = list.listIterator();
        boolean first = true;
        T previous = null;
        while (it.hasNext()) {
            T current = it.next();
            if (!first && Objects.equals(previous, current)) {
                it.remove();  // removes the element next() just returned
            }
            previous = current;
            first = false;
        }
    }

    public static void main(String[] args) {
        List<String> myList = new ArrayList<>();
        myList.add("Ace, Portgas D.");
        myList.add("Luffy, Monkey D.");
        myList.add("Luffy, Monkey D.");
        myList.add("Sanji, Vinsmoke");
        myList.add("Sanji, Vinsmoke");
        myList.add("Zoro, Roronoa");
        myList.add("Zoro, Roronoa");
        myList.add("Zoro, Roronoa");
        System.out.println("Print myList's content:");
        System.out.println(myList);
        System.out.println("After removeDuplicatesFromSortedList():");
        System.out.println(removeDuplicatesFromSortedList(myList));

        List<String> empty = new ArrayList<>();
        try {
            removeDuplicatesFromSortedList(empty);
        } catch (IndexOutOfBoundsException e) {
            System.out.println("empty list: " + e.getClass().getSimpleName());
        }
        System.out.println("dedupSorted(empty) = " + dedupSorted(empty));
        List<String> withNulls = new ArrayList<>();
        withNulls.add(null);
        withNulls.add(null);
        withNulls.add("Nami");
        System.out.println("classic with nulls: " +
removeDuplicatesFromSortedList(withNulls));
        System.out.println("dedupSorted with nulls: " + dedupSorted(withNulls));
```

Listing 13.3 (Continued) *RemoveDuplicates.java: removing adjacent duplicates from a sorted list with a remembered previous element, a version that handles empty lists and null, and an in-place version with ListIterator.*

```
        dedupSortedInPlace(myList);
        System.out.println("in place: " + myList);

        // A ListIterator can also walk backwards and replace elements.
        ListIterator<String> back = myList.listIterator(myList.size());
        StringBuilder sb = new StringBuilder("backwards:");
        while (back.hasPrevious()) {
            int index = back.previousIndex();
            String name = back.previous();
            back.set(name.toUpperCase());
            sb.append(" ").append(index).append("=").append(name.substring(0,
name.indexOf(',')));
        }
        System.out.println(sb);
        System.out.println("after set(): " + myList);
    }
}
```

```
Print myList's content:
[Ace, Portgas D., Luffy, Monkey D., Luffy, Monkey D., Sanji, Vinsmoke, Sanji, Vinsmoke,
Zoro, Roronoa, Zoro, Roronoa, Zoro, Roronoa]
After removeDuplicatesFromSortedList():
[Ace, Portgas D., Luffy, Monkey D., Sanji, Vinsmoke, Zoro, Roronoa]
empty list: IndexOutOfBoundsException
dedupSorted(empty) = []
classic with nulls: [null]
dedupSorted with nulls: [null, Nami]
in place: [Ace, Portgas D., Luffy, Monkey D., Sanji, Vinsmoke, Zoro, Roronoa]
backwards: 3=Zoro 2=Sanji 1=Luffy 0=Ace
after set(): [ACE, PORTGAS D., LUFFY, MONKEY D., SANJI, VINSMOKE, ZORO, RORONOA]
```

The first method, `removeDuplicatesFromSortedList`, is the version you will find in many textbooks, and it has two quiet defects worth studying, because both come from the same careless assumption.

- It begins with `result.add(input.get(0))`, so it assumes the list has a first element. On an empty list, `get(0)` throws `IndexOutOfBoundsException`, as the output shows.
- It uses `null` as a marker meaning "no previous element yet". The test `previous != null && !previous.equals(current)` then skips the very first element of the loop (it was already added) and adds every element that differs from its predecessor. But if the list itself contains `null`, the marker is ambiguous. With the list `[null, null, "Nami"]`, the first `null` is added by `get(0)`; then `previous` is `null` for every step, so nothing else is ever added — and `"Nami"` is lost. The comment in the code even warns that the method "can't cope with removing duplicate null values"; the truth is worse, because it drops *non-null* values too.

The second method, `dedupSorted`, fixes both defects with one idea: instead of overloading `null` with a special meaning, it keeps a separate `boolean first` that is `true` only on the first step, and it compares with `Objects.equals(previous, current)`, which treats two `null`s as equal and never throws. It works on the empty list and keeps `"Nami"`. Whenever you catch yourself using a real value such as `null`, `0` or `-1` to mean "nothing yet", ask whether that value could also occur as data; if it could, use a separate flag.

13.3.2 ListIterator: backwards, replace, insert, remove

There is one exception to the rule that iterators only move forward and only read, and it is for lists. A `List` can also hand out a `java.util.ListIterator`, a richer iterator that can move

backwards (`hasPrevious()`, `previous()`), report **indexes** (`nextIndex()`, `previousIndex()`), **replace** the element last returned (`set(e)`), **insert** at the cursor (`add(e)`), and **remove** the element last returned (`remove()`) [3].

The third method of **Listing 13.3**, `dedupSortedInPlace`, uses `it.remove()` to delete each duplicate from the list itself, so no second list is needed; `remove()` deletes the element that the most recent `next()` returned, and the iterator stays consistent because it made the change itself. The last part of `main` asks for `myList.listIterator(myList.size())` — an iterator whose cursor starts *after* the last element — and walks backwards with `hasPrevious()` and `previous()`, printing each index and replacing every name with its upper-case version through `set()`. The output shows the indexes in the order `3 2 1 0` and the list in capitals afterwards.

Python has no `ListIterator`, because it does not need one: a Python list is indexed, and the built-ins cover the rest. **Listing 13.4** does the same four jobs.

Listing 13.4 `remove_duplicates.py`: the same task in Python, with a flag, with `itertools.groupby`, in place by walking indexes from the end, and walking backwards with `reversed`.

```
"""remove_duplicates.py: removing adjacent duplicates, three ways."""
from itertools import groupby


def remove_duplicates_from_sorted_list(items):
    """The classic version: remember the previous element in a variable."""
    result = []
    first = True
    previous = None
    for current in items:
        if first or previous != current:
            result.append(current)
        previous = current
        first = False
    return result


def dedup_sorted_in_place(items):
    """In place: walk the indexes from the end so deletions do not shift what is left."""
    for i in range(len(items) - 1, 0, -1):
        if items[i] == items[i - 1]:
            del items[i]


def main():
    my_list = ["Ace, Portgas D.", "Luffy, Monkey D.", "Luffy, Monkey D.",
               "Sanji, Vinsmoke", "Sanji, Vinsmoke",
               "Zoro, Roronoa", "Zoro, Roronoa", "Zoro, Roronoa"]
    print("Print my_list's content:")
    print(my_list)
    print("After remove_duplicates_from_sorted_list():")
    print(remove_duplicates_from_sorted_list(my_list))
    print("empty list:", remove_duplicates_from_sorted_list([]))
    print("with None:", remove_duplicates_from_sorted_list([None, None, "Nami"]))

    # groupby() yields one (key, group) pair per run of equal neighbors.
    print("groupby:", [key for key, _ in groupby(my_list)])

    dedup_sorted_in_place(my_list)
    print("in place:", my_list)
```

Listing 13.4 (Continued) *remove_duplicates.py: the same task in Python, with a flag, with itertools.groupby, in place by walking indexes from the end, and walking backwards with reversed.*

```
    # Walking backwards and replacing elements: reversed() and assignment by index.
    parts = []
    for index in reversed(range(len(my_list))):
        name = my_list[index]
        my_list[index] = name.upper()   # assigning by index is always allowed
        parts.append(f"{index}={name[:name.index(',')]}")
    print("backwards:", " ".join(parts))
    print("after assignment:", my_list)


if __name__ == "__main__":
    main()
```

```
Print my_list's content:
['Ace, Portgas D.', 'Luffy, Monkey D.', 'Luffy, Monkey D.', 'Sanji, Vinsmoke', 'Sanji,
Vinsmoke', 'Zoro, Roronoa', 'Zoro, Roronoa', 'Zoro, Roronoa']
After remove_duplicates_from_sorted_list():
['Ace, Portgas D.', 'Luffy, Monkey D.', 'Sanji, Vinsmoke', 'Zoro, Roronoa']
empty list: []
with None: [None, 'Nami']
groupby: ['Ace, Portgas D.', 'Luffy, Monkey D.', 'Sanji, Vinsmoke', 'Zoro, Roronoa']
in place: ['Ace, Portgas D.', 'Luffy, Monkey D.', 'Sanji, Vinsmoke', 'Zoro, Roronoa']
backwards: 3=Zoro 2=Sanji 1=Luffy 0=Ace
after assignment: ['ACE, PORTGAS D.', 'LUFFY, MONKEY D.', 'SANJI, VINSMOKE', 'ZORO,
RORONOA']
```

`remove_duplicates_from_sorted_list` is the safe version from the start: it uses the `first` flag, and Python's `!=` compares `None` with `None` without complaint. The line with `groupby` shows a tool from the `itertools` module that we will meet again in Section 13.10: `groupby(my_list)` produces one pair `(key, group)` for every *run* of equal neighbors, so the list of keys is exactly the list without adjacent duplicates. For deleting in place, `dedup_sorted_in_place` walks the indexes **from the end towards the beginning**: deleting `items[i]` shifts only the elements *after* `i`, which the loop has already visited, so nothing is skipped. (Walking forwards while deleting is a classic mistake; Section 13.5 shows it.) Finally, `reversed(range(len(my_list)))` produces the indexes `3, 2, 1, 0`, and ordinary assignment `my_list[index] = ...` replaces an element — the job that `ListIterator.set()` does in Java.

13.4 Writing Your Own Iterable Collection

Using iterators is useful; *writing* them is where the idea pays off. Once a class of ours is iterable, it works with the for-each loop, with every library method that accepts an `Iterable`, and — in Python — with `list()`, `sum()`, `max()`, `sorted()`, `zip()` and every other function that consumes iterables.

Our example comes from the laundry floor. A customer brings in 12.5 kg of washing, and the machines take at most 5 kg per load. The attendant needs the *plan*: 5 kg, 5 kg, then 2.5 kg. We model the plan as a collection of load weights. The interesting part is that the plan never needs to *store* its loads. Given the total and the capacity, the next load can always be computed from how much is left, so the collection can produce its elements on demand.

13.4.1 Java: Iterable plus an inner iterator class

Listing 13.5 implements `Iterable<Double>` and hands out iterators of a private inner class.

Listing 13.5 `LoadPlanDemo.java`: `LoadPlan` is an `Iterable` that computes its loads on demand; each call to `iterator()` returns an independent `LoadIterator`.

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class LoadPlanDemo {
    public static void main(String[] args) {
        LoadPlan plan = new LoadPlan(12.5, 5.0);
        System.out.println(plan);
        for (double load : plan) {           // works because LoadPlan is Iterable
            System.out.println("  load of " + load + " kg");
        }

        // Every call to iterator() starts a fresh, independent walk.
        Iterator<Double> a = plan.iterator();
        Iterator<Double> b = plan.iterator();
        a.next();
        a.next();
        System.out.println("a is at its third load: " + a.next() + " kg");
        System.out.println("b is still at its first: " + b.next() + " kg");

        // Nothing is computed until someone asks: a huge plan costs nothing to create.
        LoadPlan huge = new LoadPlan(1_000_000.0, 7.5);
        Iterator<Double> h = huge.iterator();
        System.out.println("first load of the huge plan: " + h.next() + " kg");
        System.out.println("loads in the huge plan: " + huge.count());
    }
}

final class LoadPlan implements Iterable<Double> {
    private final double totalKg;
    private final double capacityKg;

    LoadPlan(double totalKg, double capacityKg) {
        if (totalKg <= 0 || capacityKg <= 0) {
            throw new IllegalArgumentException("weights must be positive");
        }
        this.totalKg = totalKg;
        this.capacityKg = capacityKg;
    }

    @Override
    public Iterator<Double> iterator() {
        return new LoadIterator();
    }
```

Listing 13.5 (Continued) *LoadPlanDemo.java: LoadPlan is an Iterable that computes its loads on demand; each call to iterator() returns an independent LoadIterator.*

```
    int count() {
        int n = 0;
        for (double ignored : this) {
            n++;
        }
        return n;
    }

    @Override
    public String toString() {
        return "LoadPlan[" + totalKg + " kg in machines of " + capacityKg + " kg]";
    }

    // An inner class: each LoadIterator can read the fields of its LoadPlan.
    private class LoadIterator implements Iterator<Double> {
        private double remaining = totalKg;  // the iterator's own position

        @Override
        public boolean hasNext() {
            return remaining > 0;
        }

        @Override
        public Double next() {
            if (!hasNext()) {
                throw new NoSuchElementException("no loads left");
            }
            double load = Math.min(remaining, capacityKg);
            remaining -= load;
            return load;
        }
    }
}
```

```
LoadPlan[12.5 kg in machines of 5.0 kg]
  load of 5.0 kg
  load of 5.0 kg
  load of 2.5 kg
a is at its third load: 2.5 kg
b is still at its first: 5.0 kg
first load of the huge plan: 7.5 kg
loads in the huge plan: 133334
```

The structure has two parts, and keeping them apart is the whole trick.

- **The collection, `LoadPlan`.** It holds only what defines the plan: `totalKg` and `capacityKg`, both `final`. Its `iterator()` does one thing: `return new LoadIterator();`. Its `count()` is a for-each loop over `this`.
- **The iterator, `LoadIterator`.** It holds only what defines a *walk*: `remaining`, the weight not yet assigned to a load, which starts at `totalKg`. `hasNext()` is `remaining > 0`. `next()` first checks `hasNext()` and throws `NoSuchElementException` if the walk is over — the contract of `Iterator` requires it — then takes `Math.min(remaining, capacityKg)` as the load, subtracts it, and returns it.

`LoadIterator` is declared `private class` *inside* `LoadPlan` without the word `static`. That makes it an **inner class**: every `LoadIterator` object is tied to the `LoadPlan` that created it and can read that plan's fields directly, which is why `remaining = totalKg` and `Math.min(remaining, capacityKg)` compile without any `plan.` prefix. Chapter 10 used inner classes for event listeners;

an iterator is the other classic use. Because the class is `private`, no one outside `LoadPlan` can even name it — the rest of the program sees only the interface `Iterator<Double>`.

The output confirms three properties. First, the for-each loop works: `for (double load : plan)` prints 5.0, 5.0 and 2.5 (the `Double` objects are unboxed to `double` automatically). Second, iterators are **independent**: after `a` has taken two loads, `a.next()` is 2.5 while a new iterator `b` still starts at 5.0; each walk has its own `remaining`. Third, the plan is **lazy**: creating a plan for a million kilograms costs nothing, the first load is available at once, and the 133,334 loads that `count()` walks through are computed one by one and never stored. (One million divided by 7.5 is $133{,}333.\overline{3}$, so there are 133,333 full loads and one final load of 2.5 kg.)

13.4.2 Python: iter, next, and generators

In Python the same design uses two special methods. An **iterable** defines `__iter__`, which returns an iterator. An **iterator** defines `__next__`, which returns the next element or raises `StopIteration`, and also `__iter__`, which returns `self` (the rule we met in Section 13.2.2). **Listing 13.6** writes both classes by hand, and then writes the same iterable a second, much shorter way.

Listing 13.6 `load_plan.py`: an iterable with a hand-written iterator class, and the same iterable written with a generator function.

```
"""load_plan.py: an iterable written by hand, then the same one as a generator."""
from itertools import islice


class LoadIterator:
    """The iterator: remembers how much is left to load."""

    def __init__(self, total_kg, capacity_kg):
        self._remaining = total_kg
        self._capacity = capacity_kg

    def __iter__(self):
        return self                     # an iterator is its own iterator

    def __next__(self):
        if self._remaining <= 0:
            raise StopIteration         # the "no more elements" signal
        load = min(self._remaining, self._capacity)
        self._remaining -= load
        return load


class LoadPlan:
    """The iterable: every call to __iter__ starts a fresh walk."""

    def __init__(self, total_kg, capacity_kg):
        if total_kg <= 0 or capacity_kg <= 0:
            raise ValueError("weights must be positive")
        self.total_kg = total_kg
        self.capacity_kg = capacity_kg

    def __iter__(self):
        return LoadIterator(self.total_kg, self.capacity_kg)

    def __repr__(self):
        return f"LoadPlan[{self.total_kg} kg in machines of {self.capacity_kg} kg]"


class GeneratorLoadPlan(LoadPlan):
    """The same iterable; a generator function writes the iterator for us."""
```

Listing 13.6 (Continued) *load_plan.py: an iterable with a hand-written iterator class, and the same iterable written with a generator function.*

```
    def __iter__(self):
        remaining = self.total_kg
        while remaining > 0:
            load = min(remaining, self.capacity_kg)
            remaining -= load
            print(f"    (generator computes {load})")
            yield load                  # pause here until the next next()


def main():
    plan = LoadPlan(12.5, 5.0)
    print(plan)
    for load in plan:
        print(f"  load of {load} kg")

    a, b = iter(plan), iter(plan)
    next(a)
    next(a)
    print("a is at its third load:", next(a), "kg")
    print("b is still at its first:", next(b), "kg")

    # An iterator is used up after one walk; the iterable is not.
    it = iter(plan)
    print("list(it) =", list(it), " list(it) again =", list(it))
    print("list(plan) twice:", list(plan), list(plan))

    lazy = GeneratorLoadPlan(12.5, 5.0)
    g = iter(lazy)
    print("generator created, nothing computed yet")
    print("first:", next(g))
    print("rest:", list(g))

    # Nothing is computed until someone asks: a huge plan costs nothing to create.
    huge = LoadPlan(1_000_000.0, 7.5)
    print("first two loads of the huge plan:", list(islice(huge, 2)))
    print("loads in the huge plan:", sum(1 for _ in huge))


if __name__ == "__main__":
    main()
```

```
LoadPlan[12.5 kg in machines of 5.0 kg]
  load of 5.0 kg
  load of 5.0 kg
  load of 2.5 kg
a is at its third load: 2.5 kg
b is still at its first: 5.0 kg
list(it) = [5.0, 5.0, 2.5]  list(it) again = []
list(plan) twice: [5.0, 5.0, 2.5] [5.0, 5.0, 2.5]
generator created, nothing computed yet
    (generator computes 5.0)
first: 5.0
    (generator computes 5.0)
    (generator computes 2.5)
rest: [5.0, 2.5]
first two loads of the huge plan: [7.5, 7.5]
loads in the huge plan: 133334
```

`LoadIterator` and `LoadPlan` mirror the Java classes line for line; the only structural difference is that Python has no inner classes with automatic access to the outer object, so `LoadPlan.__iter__` passes the two numbers to the iterator's constructor.

The class `GeneratorLoadPlan` is where Python leaves Java behind. Its `__iter__` contains the word `yield`, and a function that contains `yield` is a **generator function**: calling it does not run its body; it returns a *generator*, which is an iterator whose `__next__` runs the body *until the next* `yield`, hands over the value, and **pauses** there with all its local variables intact. The next call to `next()` resumes right after the `yield`. When the body finishes, the generator raises `StopIteration` by itself. So the eight-line `LoadIterator` class, with its constructor, its `__iter__`, its `__next__` and its explicit `StopIteration`, has become a four-line loop. Generators were added to Python for exactly this purpose by PEP 255 [7], and they have become one of the language's most characteristic tools [8].

The printed lines `(generator computes ...)` let us watch the pausing. After `g = iter(lazy)`, nothing has been computed. `next(g)` runs the body up to the first `yield` — we see one "computes" line, then `first: 5.0`. `list(g)` then drives the generator to the end, and we see the two remaining computations before the list is printed.

> **DEFINITION.** A **generator function** is a Python function whose body contains `yield`. Calling it returns a **generator**, an iterator that runs the function body step by step: each `next()` resumes the body, runs it to the next `yield`, returns the yielded value and suspends again; when the body ends, the generator raises `StopIteration`.

Java has no `yield` for iterators. The closest Java comes is a *stream* built with `Stream.iterate` or `Stream.generate`, which we meet in Section 13.10; for a class that must be `Iterable`, the explicit inner iterator of **Listing 13.5** remains the standard way.

13.4.3 An iterable is not an iterator

The middle of **Listing 13.6** demonstrates the most common confusion about Python iteration. `it = iter(plan)` is an *iterator*; `list(it)` walks it to the end and returns `[5.0, 5.0, 2.5]`; a second `list(it)` returns `[]`, because the iterator is **used up** — its cursor is past the end, and there is no way to rewind it. `list(plan)`, on the other hand, works every time, because `plan` is an *iterable*, and each `list(plan)` calls `plan.__iter__()` again and gets a fresh iterator.

Java draws the same distinction, and its type system enforces it: a for-each loop accepts an `Iterable` and refuses an `Iterator`. Python's `for` loop accepts both, because an iterator's `__iter__` returns itself. That convenience has a price: a function that loops twice over its argument works when given a list and silently does nothing the second time when given a generator. The rule of thumb is simple. If you need to walk something more than once, make sure you hold an **iterable** (a list, a tuple, an object like `LoadPlan`), not a one-shot iterator; if in doubt, call `list(...)` once and keep the list.

The last lines use `itertools.islice(huge, 2)`, which lazily takes the first two elements, and `sum(1 for _ in huge)`, which counts without building a list.

13.5 Changing a Collection While Walking It

We have been warned twice already that an iterator is not built for changing its collection. What actually happens if we try? Take the list `["a", "b", "c", "d", "e"]` and suppose an iterator has just returned `"b"`, so its cursor sits at gap 2, in front of `"c"`. Now some code removes `"a"` from the list — not through the iterator, but by calling the list's own `remove`. Every element shifts one place to the left. The iterator's cursor still says 2. What should `next()` return: `"c"`, the element that was next before the removal and is now at index 1, or `"d"`, the element that is now at index 2?

Figure 13.2 draws the dilemma. There is no good answer. Returning `"d"` skips `"c"` without a trace; returning `"c"` would require the iterator to know about every change made by anyone. Java's collections therefore refuse to guess. The iterator notices that the list has been **structurally modified** — its size changed — by someone other than itself, and the next call to `next()` throws a runtime exception of the type `ConcurrentModificationException`.

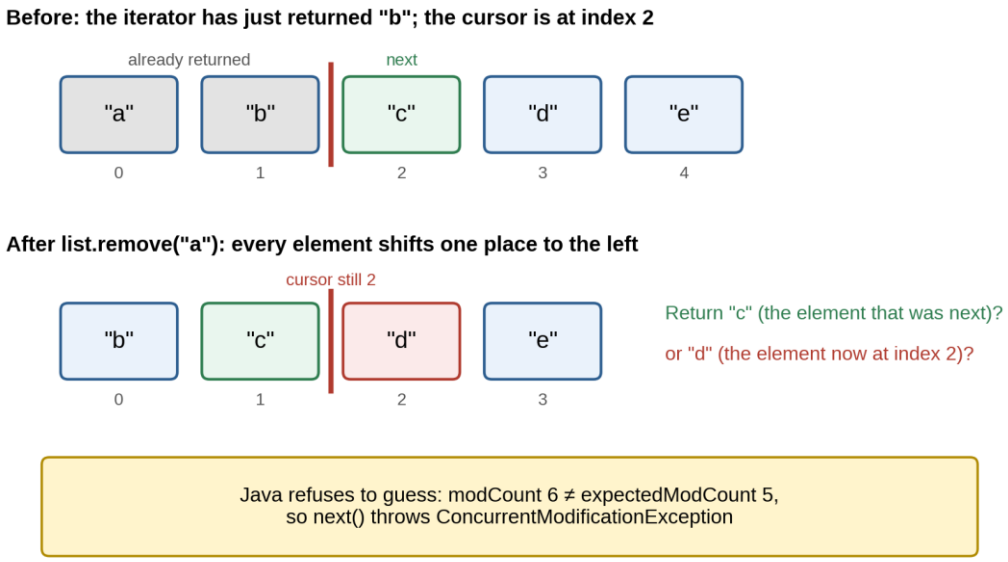


Figure 13.2 Why Java throws `ConcurrentModificationException`. After `remove("a")` the elements shift left, but the iterator's cursor does not; there is no right answer to "what comes next?", so the iterator refuses to answer.

How does the iterator notice? Every `ArrayList` keeps a counter, `modCount`, that goes up by one each time the list's size changes (every `add` and `remove`, but not `set`, which replaces an element and keeps the size). When an iterator is created, it copies the counter into its own field `expectedModCount`. Its own `remove()` updates both. And at the start of every `next()` it checks

$$\text{modCount} = \text{expectedModCount} \tag{13.2}$$

If Equation (13.2) is false, someone else changed the list, and the iterator throws [3]. The name is a little misleading: "concurrent" here does not mean two threads (Chapter 14 is about those); it means that the modification happened *during* the iteration, even in a single thread.

The word *structurally* explains a detail of the rules: while iterating over a `List`, calling `add` or `remove` on the list is forbidden, but calling `set` is allowed, because `set` does not move any element and does not change `modCount`. **Listing 13.7** tries all three, and then shows the correct ways.

Listing 13.7 `ConcurrentModification.java`: `set` during iteration is allowed; `add` and `remove` through the list throw `ConcurrentModificationException`; one case slips through silently; three correct alternatives.

```
import java.util.ArrayList;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.List;

public class ConcurrentModification {
    static List<String> crew() {
        return new ArrayList<>(List.of("Ace", "Luffy", "Sanji", "Zoro", "Nami"));
    }

    public static void main(String[] args) {
        // 1. set() in the middle of iterating: allowed, the size does not change.
        List<String> myList = crew();
        for (String name : myList) {
            if (name.equals("Luffy")) {
                myList.set(2, "Sabo");        // replace Sanji, who is not yet visited
            }
            System.out.print(name + " ");
        }
        System.out.println(" <- set() is fine");

        // 2. add() in the middle of iterating: the iterator notices.
        myList = crew();
        try {
            for (String name : myList) {
                if (name.equals("Luffy")) {
                    myList.add("Buggy");
                }
            }
        } catch (ConcurrentModificationException e) {
            System.out.println("add() while iterating: " + e.getClass().getSimpleName());
        }

        // 3. remove() through the list, not through the iterator: same verdict.
        myList = crew();
        try {
            for (String name : myList) {
                if (name.equals("Ace")) {
                    myList.remove(name);
                }
            }
        } catch (ConcurrentModificationException e) {
            System.out.println("remove() while iterating: " +
e.getClass().getSimpleName());
        }
```

Listing 13.7 (Continued) *ConcurrentModification.java: set during iteration is allowed; add and remove through the list throw ConcurrentModificationException; one case slips through silently; three correct alternatives.*

```
        // 4. The check is best effort: removing the second-to-last element
        //    ends the loop early, silently, and Nami is never visited.
        myList = crew();
        StringBuilder seen = new StringBuilder();
        for (String name : myList) {
            seen.append(name).append(' ');
            if (name.equals("Zoro")) {
                myList.remove(name);
            }
        }
        System.out.println("no exception, but visited only: " + seen.toString().trim());

        // 5. The right ways.
        myList = crew();
        Iterator<String> it = myList.iterator();
        while (it.hasNext()) {
            if (it.next().length() <= 3) {
                it.remove();                  // removal through the iterator itself
            }
        }
        System.out.println("Iterator.remove(): " + myList);

        myList = crew();
        myList.removeIf(name -> name.length() <= 3);
        System.out.println("removeIf():        " + myList);

        myList = crew();
        for (String name : new ArrayList<>(myList)) {  // iterate over a copy
            if (name.startsWith("S")) {
                myList.add(name + " (again)");
            }
        }
        System.out.println("copy, then add:    " + myList);
    }
}
```

```
Ace Luffy Sabo Zoro Nami  <- set() is fine
add() while iterating: ConcurrentModificationException
remove() while iterating: ConcurrentModificationException
no exception, but visited only: Ace Luffy Sanji Zoro
Iterator.remove(): [Luffy, Sanji, Zoro, Nami]
removeIf():        [Luffy, Sanji, Zoro, Nami]
copy, then add:    [Ace, Luffy, Sanji, Zoro, Nami, Sanji (again)]
```

The first three blocks behave as the rules predict. Block 1 replaces `Sanji` with `Sabo` while the loop is at `Luffy`, and the loop then visits `Sabo` normally. Blocks 2 and 3 change the size of the list inside a for-each loop, and the *next* call to `next()` — hidden in the loop header — throws.

Block 4 is the reason the Java documentation insists that this behavior is only a "best-effort" check that "should not be relied upon" [3]. The loop removes `Zoro`, the second-to-last element, when it reaches him. Now apply Equation (13.1). After `next()` returned `Zoro`, the cursor c was 4. The removal shrank the list from 5 to 4 elements, so $n = 4$. The hidden `hasNext()` asks whether $c \neq n$, which is $4 \neq 4$: false. The loop simply *ends* — before `next()` ever gets a chance to check Equation (13.2) — and `Nami` is never visited. No exception, no warning, just a wrong answer. That is why the rule is not "catch `ConcurrentModificationException`" but "**never** change a collection's size while a for-each loop walks it".

Block 5 shows three safe ways to do what blocks 2–4 attempted:

- **Remove through the iterator.** Write the explicit `while (it.hasNext())` loop and call `it.remove()`. The iterator removes the element it last returned and updates its own `expectedModCount`, so everything stays consistent.
- **Use `removeIf`.** Since Java 8, every collection has `removeIf(predicate)`, which removes all elements for which the condition is true. `myList.removeIf(name -> name.length() <= 3)` says what we mean in one line, and the library does the iterator work correctly.
- **Iterate over a copy.** `for (String name : new ArrayList<>(myList))` walks a snapshot, so the real list may be changed freely inside the loop. This costs a copy, but it is the simplest choice when you must *add* elements during the walk.

Python lists take a different attitude: they do **not** check. **Listing 13.8** shows what that means.

Listing 13.8 `concurrent_modification.py`: a Python list changed during a `for` loop skips elements silently; dictionaries and sets raise `RuntimeError`; three correct alternatives.

```
"""concurrent_modification.py: changing a collection while walking over it."""


def crew():
    return ["Ace", "Luffy", "Sanji", "Zoro", "Nami"]


def main():
    # 1. Replacing an element by index while iterating: allowed.
    my_list = crew()
    for name in my_list:
        if name == "Luffy":
            my_list[2] = "Sabo"
        print(name, end=" ")
    print(" <- assignment is fine")

    # 2. Removing from a list while iterating: no error, just wrong answers.
    numbers = [2, 4, 6, 7]
    for n in numbers:
        if n % 2 == 0:
            numbers.remove(n)
    print("remove evens while iterating:", numbers)

    # 3. The same mistake with the crew: Nami is never visited.
    my_list = crew()
    seen = []
    for name in my_list:
        seen.append(name)
        if name == "Zoro":
            my_list.remove(name)
    print("visited only:", seen)

    # 4. Dictionaries and sets do check, and raise RuntimeError.
    stock = {"soap": 3, "softener": 0, "bleach": 1}
    try:
        for item in stock:
            if stock[item] == 0:
                del stock[item]
    except RuntimeError as e:
        print("dict:", e)
    tags = {"wash", "iron"}
    try:
        for t in tags:
            tags.add(t.upper())
    except RuntimeError as e:
        print("set:", e)
```

Listing 13.8 (Continued) *concurrent_modification.py: a Python list changed during a for loop skips elements silently; dictionaries and sets raise RuntimeError; three correct alternatives.*

```
    # 5. The right ways.
    numbers = [2, 4, 6, 7]
    numbers = [n for n in numbers if n % 2 != 0]       # build a new list
    print("comprehension:", numbers)

    my_list = crew()
    my_list[:] = [name for name in my_list if len(name) > 3]  # replace contents in place
    print("slice assignment:", my_list)

    stock = {"soap": 3, "softener": 0, "bleach": 1}
    for item in list(stock):                           # iterate over a copy of the keys
        if stock[item] == 0:
            del stock[item]
    print("copy of the keys:", stock)


if __name__ == "__main__":
    main()
```

```
Ace Luffy Sabo Zoro Nami  <- assignment is fine
remove evens while iterating: [4, 7]
visited only: ['Ace', 'Luffy', 'Sanji', 'Zoro']
dict: dictionary changed size during iteration
set: Set changed size during iteration
comprehension: [7]
slice assignment: ['Luffy', 'Sanji', 'Zoro', 'Nami']
copy of the keys: {'soap': 3, 'bleach': 1}
```

Block 2 is the famous trap. We want to remove the even numbers from `[2, 4, 6, 7]`, and the answer should be `[7]`. The list iterator keeps an index. At index 0 it sees 2 and removes it; the list becomes `[4, 6, 7]`; the index moves to 1, which now holds **6** — so **4 was skipped**. 6 is removed, the list becomes `[4, 7]`, the index moves to 2, which is past the end, and the loop stops. The result `[4, 7]` is wrong, and Python said nothing. Block 3 is the same mistake with names: removing `Zoro` makes `Nami` move into the position the loop has already passed.

Dictionaries and sets are stricter. Changing their size while iterating over them raises `RuntimeError` with the message `dictionary changed size during iteration` or `Set changed size during iteration`, because a hash table that grows or shrinks may rearrange its elements completely (Section 13.9 explains why), and continuing the walk would be meaningless.

The Python remedies in block 5 mirror the Java ones: build a **new list** with a comprehension (Section 13.10 explains the syntax); replace a list's contents **in place** with slice assignment `my_list[:] = [...]`, which keeps the same list object for anyone else who holds a reference to it; or iterate over a **copy** such as `list(stock)`, the list of the dictionary's keys taken before the loop starts. **Table 13.2** summarizes the safe patterns.

Table 13.2 Safe ways to change a collection that is being walked.

Goal while walking	Java	Python
Replace an element	`list.set(i, e)` or `ListIterator.set(e)`	`lst[i] = e`
Remove matching elements	`it.remove()` in an explicit loop, or `c.removeIf(p)`	`lst[:] = [x for x in lst if keep(x)]`
Add elements	loop over `new ArrayList<>(list)`	loop over `list(lst)` or `lst[:]`
Delete dictionary entries	`map.entrySet().removeIf(p)`	loop over `list(d)` and `del d[k]`
What happens if you break the rule	usually `ConcurrentModificationException`; sometimes a silent skip	lists: silent skips; dicts and sets: `RuntimeError`

13.6 Putting Objects in Order: Comparable and Comparator

Lists are ordered, and it is often useful to sort their elements. Java's `Collections.sort(list)` (or `list.sort(...)`) and Python's `list.sort()` and `sorted(...)` do the work, but they need to know *how to compare two elements*. Not every list can be sorted: a list of strings can, because strings have an alphabetical order, but what is the order of a list of students?

Java gives a class two ways to answer. If the class has one **natural order** that everyone would agree on — numbers by size, strings alphabetically, dates by time — it implements the interface `Comparable<T>`, whose single method `int compareTo(T other)` compares `this` with `other`. Chapter 9 used exactly this interface as the bound in the generic signature of sorting:

```
public static <E extends Comparable<E>> void sort(List<E> list)
```

which reads "`sort` accepts a list of any type `E`, provided that `E` knows how to compare itself with other `E`s". (The library's real bound, `<T extends Comparable<? super T>>`, is slightly more general, for the reasons of Chapter 9's wildcards.)

But many classes have no single natural order. Each student has a first name, a last name, an age, a degree course, marks and so on. How should a list of students be sorted? It depends on what information you need from that list: by name for a class register, by age for a sports team, by marks for a prize. Building one of these into the class with `Comparable` would confuse more than it helps. This is why the second interface exists: a **`Comparator<T>`** is a *separate* object whose method `int compare(T a, T b)` compares two objects from outside [3]. A class may have one natural order and any number of comparators.

13.6.1 The contract every comparison must keep

Both `compareTo` and `compare` return an `int` whose **sign** is the answer: negative means "the first comes before the second", zero means "they are equal in this order", positive means "the first comes after". Only the sign matters; `-1` and `-37` mean the same thing. Sorting algorithms rely on three rules, which the documentation calls the general contract. Writing $\text{sgn}(x)$ for the sign of x (-1, 0 or 1) and cmp for the comparison, the rules are:

$$\text{sgn}(\text{cmp}(a, b)) = -\text{sgn}(\text{cmp}(b, a)) \quad \text{for all } a, b \tag{13.3}$$

$$\text{cmp}(a, b) > 0 \text{ and } \text{cmp}(b, c) > 0 \;\Rightarrow\; \text{cmp}(a, c) > 0 \tag{13.4}$$

and, third, if $\mathrm{cmp}(a, b) = 0$ then a and b compare the same way with every other element. Equation (13.3) says the order is *antisymmetric*: if a comes before b, then b comes after a. Equation (13.4) says it is *transitive*: if a is after b and b is after c, then a is after c. These are the properties you take for granted when you put things in a line, and a comparison that breaks them makes the sort's result meaningless [9].

A common shortcut breaks them. To compare two ages, it is tempting to return `s1.getAge() - s2.getAge()`: the difference is negative, zero or positive exactly when it should be. For ages that is harmless. But an `int` in Java holds only values from -2^{31} to $2^{31} - 1$, and a difference of two large values can *overflow*:

$$2^{31} - 1 \;-\; (-1) \;=\; 2^{31} \;\overset{\text{overflow}}{\rightarrow}\; -2^{31} < 0 \tag{13.5}$$

Equation (13.5) shows the largest `int` compared with -1: the true difference is positive, but the computed one wraps around to the most negative `int`, so the comparison claims the largest number is *smaller* than -1, breaking Equation (13.3). The cure costs nothing: `Integer.compare(a, b)` (and `Long.compare`, `Double.compare`) returns the correct sign for every pair of values.

13.6.2 A comparator by age

Listing 13.9 sorts three students by age with a hand-written `AgeComparator`, then with comparators built from library parts.

Listing 13.9 `AgeSort.java`: a hand-written `AgeComparator` that also handles `null`, and the same orders built with `Comparator.comparingInt`, `reversed` and `nullsFirst`.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;

public class AgeSort {
    static final class Student {
        private final String name;
        private final int age;

        Student(String name, int age) {
            this.name = name;
            this.age = age;
        }

        String getName() {
            return name;
        }

        int getAge() {
            return age;
        }

        @Override
        public String toString() {
            return name + " (" + age + ")";
        }
    }

    // A comparator is a separate object that knows how to order two students.
    static final class AgeComparator implements Comparator<Student> {
        @Override
        public int compare(Student s1, Student s2) {
            if (s1 == s2) {
                return 0;            // the same object, or both null
            } else if (s1 == null) {
                return -1;           // s1 should come before s2
            } else if (s2 == null) {
                return 1;            // s1 should come after s2
            } else {
                return Integer.compare(s1.getAge(), s2.getAge());
            }
        }
    }
```

Listing 13.9 (Continued) *AgeSort.java: a hand-written AgeComparator that also handles null, and the same orders built with Comparator.comparingInt, reversed and nullsFirst.*

```
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        Student s = new Student("Chopper", 15);
        students.add(s);
        Student s1 = new Student("Nami", 20);
        students.add(s1);
        Student s2 = new Student("Usopp", 12);
        students.add(s2);
        Iterator<Student> itStudents = students.iterator();
        System.out.println("Print the list of student:");
        while (itStudents.hasNext()) {
            System.out.println(itStudents.next().getAge());
        }
        AgeComparator age = new AgeComparator();
        System.out.println();
        System.out.printf("Age comparison between s1 (age %d) and s2 (age %d) = %d%n",
                s1.getAge(), s2.getAge(), age.compare(s1, s2));
        System.out.println();
        System.out.println("After sort is called, the list of student becomes:");
        Collections.sort(students, age);
        for (Student st : students) {
            System.out.println(st.getAge());
        }

        // The same orders, built from small pieces instead of a whole class.
        students.sort(Comparator.comparingInt(Student::getAge).reversed());
        System.out.println("oldest first: " + students);
        students.add(null);
        students.sort(Comparator.nullsFirst(Comparator.comparing(Student::getName)));
        System.out.println("by name, nulls first: " + students);
    }
}
```

```
Print the list of student:
15
20
12

Age comparison between s1 (age 20) and s2 (age 12) = 1

After sort is called, the list of student becomes:
12
15
20
oldest first: [Nami (20), Chopper (15), Usopp (12)]
by name, nulls first: [null, Chopper (15), Nami (20), Usopp (12)]
```

`AgeComparator.compare` treats the special cases first. `s1 == s2` covers both "the very same object" and "both are `null`", and returns 0. If only `s1` is `null`, it returns -1 so that `null`s come first; if only `s2` is `null`, it returns 1. Otherwise it returns `Integer.compare(s1.getAge(), s2.getAge())`. For Nami (20) and Usopp (12), that is 1: Nami comes after Usopp in age order. (The subtraction shortcut would have returned $20 - 12 = 8$, with the same sign and, for ages, the same effect.) `Collections.sort(students, age)` then puts the ages in the order 12, 15, 20.

The last lines show the modern style. `Comparator.comparingInt(Student::getAge)` builds the same comparator from a *key extractor* — a method reference that says which value to compare — and `.reversed()` flips it for "oldest first". `Comparator.nullsFirst(Comparator.comparing(Student::getName))` compares by name and puts `null` in front, which is the `null` handling of `AgeComparator` in one call. These combinators exist precisely so that nobody has to write the error-prone parts by hand.

Python's sort takes a different route. Instead of a comparison function that looks at *two* elements, Python's `sort` and `sorted` take a **key function** that looks at *one* element and returns the value to sort by. The sort computes each key once and then compares the keys with `<`. **Listing 13.10** shows both styles.

Listing 13.10 `age_sort.py`: sorting with a key function (`attrgetter`, `lambda`), `reverse=True`, and a Java-style comparator adapted with `functools.cmp_to_key`.

```
"""age_sort.py: ordering objects with key functions, and with a comparator."""
from functools import cmp_to_key
from operator import attrgetter


class Student:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    @property
    def name(self):
        return self._name

    @property
    def age(self):
        return self._age

    def __repr__(self):
        return f"{self._name} ({self._age})"


def age_compare(s1, s2):
    """A Java-style comparator: negative, zero or positive."""
    if s1 is s2:
        return 0
    if s1 is None:
        return -1
    if s2 is None:
        return 1
    return (s1.age > s2.age) - (s1.age < s2.age)


def main():
    s = Student("Chopper", 15)
    s1 = Student("Nami", 20)
    s2 = Student("Usopp", 12)
    students = [s, s1, s2]
    print("Print the list of student:")
    for st in students:
        print(st.age)
    print()
    print(f"Age comparison between s1 (age {s1.age}) and s2 (age {s2.age}) = "
          f"{age_compare(s1, s2)}")
    print()
    print("After sort is called, the list of student becomes:")
```

Listing 13.10 (Continued) *age_sort.py: sorting with a key function (attrgetter, lambda), reverse=True, and a Java-style comparator adapted with functools.cmp_to_key.*

```
    students.sort(key=attrgetter("age"))       # Python's usual way: a key function
    for st in students:
        print(st.age)

    students.sort(key=lambda st: st.age, reverse=True)
    print("oldest first:", students)
    students.append(None)
    students.sort(key=cmp_to_key(lambda a, b: -1 if a is None else 1 if b is None
                                 else (a.name > b.name) - (a.name < b.name)))
    print("by name, None first:", students)


if __name__ == "__main__":
    main()
```

```
Print the list of student:
15
20
12

Age comparison between s1 (age 20) and s2 (age 12) = 1

After sort is called, the list of student becomes:
12
15
20
oldest first: [Nami (20), Chopper (15), Usopp (12)]
by name, None first: [None, Chopper (15), Nami (20), Usopp (12)]
```

`students.sort(key=attrgetter("age"))` sorts by the attribute `age`; `operator.attrgetter("age")` is a ready-made function equivalent to `lambda st: st.age`. `reverse=True` replaces Java's `.reversed()`. When you really have a two-argument comparison — perhaps ported from Java — `functools.cmp_to_key` wraps it into a key; the last sort uses one to put `None` first. The expression `(a > b) - (a < b)` is Python's idiom for "the sign of the comparison", because `True` and `False` count as 1 and 0.

Key functions have two advantages. They are harder to get wrong: a key cannot break Equation (13.3) or Equation (13.4), because the comparison of the keys is done by Python's own `<`. And they are faster: each key is computed once per element rather than once per comparison. Python's sort is also **stable**: elements with equal keys keep their original relative order. Java's sort of objects is stable too. Stability lets you sort by several criteria in several passes — sort by first name, then (stably) by last name, and people with the same last name remain ordered by first name [10].

13.6.3 Names in three orders, and a comparison that only looks right

The next example sorts five members of one famous fictional family by full name in several orders, each for two different starting orders ("Case 1" and "Case 2"). A correct sort must give the *same* result for both cases, since the result depends only on the names, not on where they started. **Listing 13.11** includes, deliberately, a `compareTo` of a kind that is written surprisingly often.

Listing 13.11 `NameOrder.java`: a `compareTo` that handles only −1, 0 and 1, a correct natural order, a last-then-first comparator, and why comparing glued strings is fragile.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class NameOrder {
    // A natural order that looks right but is not: compareTo may return any int.
    static final class BrokenName implements Comparable<BrokenName> {
        final String firstName, lastName;

        BrokenName(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }

        @Override
        public int compareTo(BrokenName other) {
            int result = Integer.MIN_VALUE;
            switch (firstName.compareTo(other.firstName)) {
                case -1:
                    result = -1;
                    break;
                case 0:
                    result = lastName.compareTo(other.lastName);
                    break;
                case 1:
                    result = 1;
                    break;
            }
            return result;
        }

        @Override
        public String toString() {
            return firstName + " " + lastName.charAt(0) + ".";  // e.g. "Luffy M."
        }
    }

    // The same idea, done right: pass the sign on, whatever its size.
    record Name(String firstName, String lastName) implements Comparable<Name> {
        @Override
        public int compareTo(Name other) {
            int byFirst = firstName.compareTo(other.firstName);
            return byFirst != 0 ? byFirst : lastName.compareTo(other.lastName);
        }
```

Listing 13.11 (Continued) *NameOrder.java: a compareTo that handles only -1, 0 and 1, a correct natural order, a last-then-first comparator, and why comparing glued strings is fragile.*

```
        @Override
        public String toString() {
            return firstName + " " + lastName.charAt(0) + ".";
        }
    }

    static final Comparator<Name> LAST_THEN_FIRST =
            Comparator.comparing(Name::lastName).thenComparing(Name::firstName);

    // Glues the two names together and compares the glued strings.
    static final Comparator<Name> CONCATENATED =
            (n1, n2) -> (n1.lastName() + n1.firstName()).compareTo(n2.lastName() +
n2.firstName());

    static String show(List<?> items) {
        List<String> parts = new ArrayList<>();
        for (Object item : items) {
            parts.add(item.toString());
        }
        return String.join(", ", parts);
    }

    public static void main(String[] args) {
        String[][] case1 = {{"Luffy", "Monkey D."}, {"Garp", "Monkey D."},
            {"Dragon", "Monkey D."}, {"Rouge", "Portgas D."}, {"Ace", "Portgas D."}};
        String[][] case2 = {{"Luffy", "Monkey D."}, {"Rouge", "Portgas D."},
            {"Garp", "Monkey D."}, {"Ace", "Portgas D."}, {"Dragon", "Monkey D."}};
        System.out.println("\"Luffy\".compareTo(\"Garp\") = " + "Luffy".compareTo("Garp"));

        int caseNo = 1;
        for (String[][] data : List.of(case1, case2)) {
            BrokenName[] broken = new BrokenName[data.length];
            List<Name> names = new ArrayList<>();
            for (int i = 0; i < data.length; i++) {
                broken[i] = new BrokenName(data[i][0], data[i][1]);
                names.add(new Name(data[i][0], data[i][1]));
            }
            System.out.println("Case " + caseNo++);
            System.out.println("  original order:   " + show(names));
            Arrays.sort(broken);
            System.out.println("  broken compareTo: " + show(Arrays.asList(broken)));
            names.sort(null);                       // null means: use compareTo
            System.out.println("  first, last:      " + show(names));
            names.sort(LAST_THEN_FIRST);
            System.out.println("  last, first:      " + show(names));
        }
```

Listing 13.11 (Continued) *NameOrder.java: a compareTo that handles only -1, 0 and 1, a correct natural order, a last-then-first comparator, and why comparing glued strings is fragile.*

```
        // Why gluing strings together is fragile: a space sorts before a letter.
        List<Name> tricky = new ArrayList<>(List.of(
                new Name("Luffy", "Monkey D."), new Name("Zed", "Monkey")));
        tricky.sort(CONCATENATED);
        System.out.println("concatenated:     " + tricky.get(0).lastName() + " first");
        tricky.sort(LAST_THEN_FIRST);
        System.out.println("last, then first: " + tricky.get(0).lastName() + " first");
    }
}
```

```
"Luffy".compareTo("Garp") = 5
Case 1
  original order:   Luffy M., Garp M., Dragon M., Rouge P., Ace P.
  broken compareTo: Ace P., Rouge P., Dragon M., Garp M., Luffy M.
  first, last:      Ace P., Dragon M., Garp M., Luffy M., Rouge P.
  last, first:      Dragon M., Garp M., Luffy M., Ace P., Rouge P.
Case 2
  original order:   Luffy M., Rouge P., Garp M., Ace P., Dragon M.
  broken compareTo: Dragon M., Ace P., Garp M., Rouge P., Luffy M.
  first, last:      Ace P., Dragon M., Garp M., Luffy M., Rouge P.
  last, first:      Dragon M., Garp M., Luffy M., Ace P., Rouge P.
concatenated:     Monkey D. first
last, then first: Monkey first
```

Look at `BrokenName.compareTo`. It compares the first names with `String.compareTo` and then uses a `switch` with the cases `-1`, `0` and `1`, returning `Integer.MIN_VALUE` for anything else. Whoever writes such code believes that `compareTo` returns only -1, 0 or 1. It does not. The first line of the output shows `"Luffy".compareTo("Garp") = 5`: `String.compareTo` returns the difference between the first two characters that differ, here 'L' $-$ 'G' $= 76 - 71 = 5$. So for almost every pair of names the `switch` falls through to `Integer.MIN_VALUE`, a negative number meaning "less" — whatever the order of the arguments. The comparison says $a < b$ *and* $b < a$, which breaks Equation (13.3).

What does the sort do with such a comparison? Java's sort for objects is a *Timsort*, which begins by looking for a **run** — a stretch of the array that is already in order. It asks whether the second element is less than the first; if so, it keeps extending a *strictly descending* run as long as each element is "less" than the one before, and then reverses that run in place. Since the broken comparison answers "less" to every question, the whole array looks like one long descending run, and the "sorted" array is simply the **input reversed**. Compare the output: Case 1 started as Luffy, Garp, Dragon, Rouge, Ace, and "broken compareTo" printed Ace, Rouge, Dragon, Garp, Luffy — exactly backwards. Case 2 started in a different order and came out as a different (again reversed) order. That is the telltale sign of a broken comparison: **the result depends on the input order**. On larger arrays, Java's sort may even detect the inconsistency and throw `IllegalArgumentException: Comparison method violates its general contract!`.

The correct `Name` record fixes it in the simplest way: it passes the result of the first comparison on, whatever its size, and only when it is zero compares the last names. Both cases now give Ace, Dragon, Garp, Luffy, Rouge. `LAST_THEN_FIRST` orders by last name and then by first name with `Comparator.comparing(...).thenComparing(...)`, giving the Monkey D. family before the Portgas D. family, each sorted by first name, again identically for both cases.

The last lines address a tempting shortcut: glue last name and first name into one string and compare the glued strings. That usually works, but not always. `"Monkey D." + "Luffy"` is `"Monkey D.Luffy"` and `"Monkey" + "Zed"` is `"MonkeyZed"`. They agree on the first six letters; then a space (character 32) meets a `Z` (character 90), and the space is smaller, so the glued comparison puts Luffy of the "Monkey D." family first — although the last name `"Monkey"` is a prefix of `"Monkey D."` and therefore comes first in a correct last-then-first order. Comparing field by field never has this problem.

Listing 13.12 does the same in Python.

Listing 13.12 `name_order.py`: `@dataclass(order=True)` for a field-by-field natural order, tuple keys for other orders, and the broken comparison reproduced with `cmp_to_key`.

```
"""name_order.py: sorting names by several keys, and a broken comparator."""
from dataclasses import dataclass
from functools import cmp_to_key


@dataclass(frozen=True, order=True)       # order=True compares field by field
class Name:
    first_name: str
    last_name: str

    def __str__(self):
        return f"{self.first_name} {self.last_name[0]}."


def java_compare(a, b):
    """Java's String.compareTo: difference of the first differing characters."""
    for x, y in zip(a, b):
        if x != y:
            return ord(x) - ord(y)
    return len(a) - len(b)


def broken_compare(n1, n2):
    """The same mistake as Java's BrokenName: only -1, 0 and 1 are handled."""
    result = -(2 ** 31)
    c = java_compare(n1.first_name, n2.first_name)
    if c == -1:
        result = -1
    elif c == 0:
        result = java_compare(n1.last_name, n2.last_name)
    elif c == 1:
        result = 1
    return result


def show(items):
    return ", ".join(str(x) for x in items)


def main():
    case1 = [("Luffy", "Monkey D."), ("Garp", "Monkey D."), ("Dragon", "Monkey D."),
             ("Rouge", "Portgas D."), ("Ace", "Portgas D.")]
    case2 = [("Luffy", "Monkey D."), ("Rouge", "Portgas D."), ("Garp", "Monkey D."),
             ("Ace", "Portgas D."), ("Dragon", "Monkey D.")]
    print('java_compare("Luffy", "Garp") =', java_compare("Luffy", "Garp"))
```

Listing 13.12 (Continued) *name_order.py: @dataclass(order=True) for a field-by-field natural order, tuple keys for other orders, and the broken comparison reproduced with cmp_to_key.*

```
    for case_no, data in enumerate([case1, case2], start=1):
        names = [Name(f, l) for f, l in data]
        print(f"Case {case_no}")
        print("  original order:  ", show(names))
        print("  broken compare:  ", show(sorted(names, key=cmp_to_key(broken_compare))))
        print("  first, last:     ", show(sorted(names)))
        print("  last, first:     ",
              show(sorted(names, key=lambda n: (n.last_name, n.first_name))))

    tricky = [Name("Luffy", "Monkey D."), Name("Zed", "Monkey")]
    glued = sorted(tricky, key=lambda n: n.last_name + n.first_name)
    print("concatenated:    ", glued[0].last_name, "first")
    proper = sorted(tricky, key=lambda n: (n.last_name, n.first_name))
    print("last, then first:", proper[0].last_name, "first")


if __name__ == "__main__":
    main()
```

```
java_compare("Luffy", "Garp") = 5
Case 1
  original order:   Luffy M., Garp M., Dragon M., Rouge P., Ace P.
  broken compare:   Ace P., Rouge P., Dragon M., Garp M., Luffy M.
  first, last:      Ace P., Dragon M., Garp M., Luffy M., Rouge P.
  last, first:      Dragon M., Garp M., Luffy M., Ace P., Rouge P.
Case 2
  original order:   Luffy M., Rouge P., Garp M., Ace P., Dragon M.
  broken compare:   Dragon M., Ace P., Garp M., Rouge P., Luffy M.
  first, last:      Ace P., Dragon M., Garp M., Luffy M., Rouge P.
  last, first:      Dragon M., Garp M., Luffy M., Ace P., Rouge P.
concatenated:     Monkey D. first
last, then first: Monkey first
```

`@dataclass(frozen=True, order=True)` asks Python to generate `<`, `<=`, `>` and `>=` that compare the fields in the order they are declared, first `first_name` and then `last_name` — exactly the correct `compareTo` of the Java record, written by the library. For other orders we return a **tuple** from the key function: `key=lambda n: (n.last_name, n.first_name)`. Tuples compare field by field, left to right, so a tuple key is the Python spelling of `comparing(...).thenComparing(...)`. To reproduce the broken comparison faithfully, `java_compare` imitates `String.compareTo` (Python's own string comparison returns only `True` or `False`), and `cmp_to_key(broken_compare)` feeds it to `sorted`. The result is the same reversed input, for the same reason: Python's sort is the original Timsort, and it, too, begins by detecting a descending run and reversing it [10].

13.7 Copies: Aliasing, Shallow and Deep

When programming, it is often useful to create a copy of an object — a list of people, say — so that one part of the program can change the copy without disturbing the original. With lists of objects, copying raises a question that surprises almost every beginner: **how deep does the copy go?**

Consider this sequence. We build a list `people` holding two `Person` objects, Usopp and Nami; we make `copy` from it; we take the first person of the copy and rename him Brook. What does `copy` look like? Obviously `[Brook, Nami]`. What does the *original* list look like? Most people would expect `[Usopp, Nami]`. But `[Brook, Nami]` is equally possible — it depends entirely on what kind of copy was made. There are three levels, and **Figure 13.3** draws them.

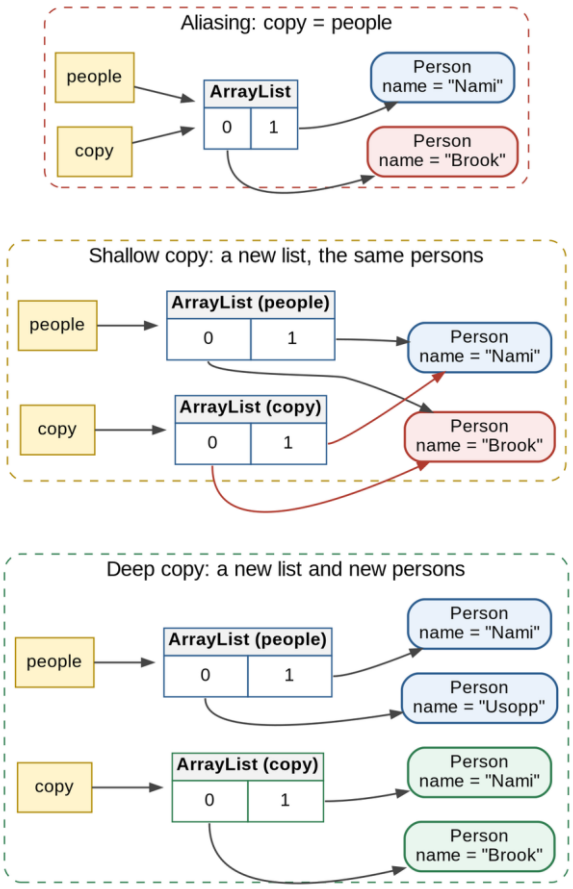


Figure 13.3 Three kinds of "copy" of a list of persons. Aliasing: two variables, one list. Shallow copy: two lists, but their slots refer to the same `Person` objects, so renaming through one list is visible through the other. Deep copy: two lists and separate `Person` objects.

> **DEFINITION. Aliasing** is having two references to the same object; nothing is copied. A **shallow copy** of a collection is a new collection object that holds the *same element references* as the original: the order of the objects is copied, but the objects themselves are shared. A **deep copy** copies the collection *and* every element (and, recursively, whatever the elements refer to), so that the copy shares nothing mutable with the original.

Listing 13.13 runs the same experiment at all three levels. In each experiment, it checks with `==` whether the two lists and their first elements are the same objects, renames the first person of the copy to Brook, adds Franky to the copy only, and prints both lists.

Listing 13.13 `CopyDemo.java`: the same rename-and-add experiment on an alias, a shallow copy and a deep copy of a list of `Person` objects.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CopyDemo {
    static final class Person {
        private String name;

        Person(String name) {
            this.name = name;
        }

        String getName() {
            return name;
        }

        void setName(String name) {
            this.name = name;
        }

        @Override
        public String toString() {
            return "[" + name + "]";
        }
    }

    // Shallow: a new list, but the same Person objects inside it.
    private static List<Person> copyList(List<Person> people) {
        List<Person> result = new ArrayList<Person>(people);
        Collections.copy(result, people);   // copies references, not persons
        return result;
    }

    // Deep: a new list AND a new Person for every element.
    private static List<Person> deepCopy(List<Person> people) {
        List<Person> result = new ArrayList<>(people);
        for (int i = 0; i < people.size(); i++) {
            result.set(i, new Person(people.get(i).getName()));
        }
        return result;
    }
```

Listing 13.13 (Continued) *CopyDemo.java: the same rename-and-add experiment on an alias, a shallow copy and a deep copy of a list of Person objects.*

```
    static List<Person> crew() {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Usopp"));
        people.add(new Person("Nami"));
        return people;
    }

    static void experiment(String title, List<Person> people, List<Person> copy) {
        System.out.println(title);
        System.out.println("  same list object?   " + (copy == people));
        System.out.println("  same first person?  " + (copy.get(0) == people.get(0)));
        copy.get(0).setName("Brook");        // Update the copied list: Usopp -> Brook
        copy.add(new Person("Franky"));      // and add someone to the copy only
        System.out.println("  the copied list:    " + copy);
        System.out.println("  the original list:  " + people);
    }

    public static void main(String[] args) {
        List<Person> people = crew();
        experiment("Aliasing (no copy at all)", people, people);
        people = crew();
        experiment("Shallow copying", people, copyList(people));
        people = crew();
        experiment("Deep copying", people, deepCopy(people));
    }
}
```

```
Aliasing (no copy at all)
  same list object?   true
  same first person?  true
  the copied list:    [[Brook], [Nami], [Franky]]
  the original list:  [[Brook], [Nami], [Franky]]
Shallow copying
  same list object?   false
  same first person?  true
  the copied list:    [[Brook], [Nami], [Franky]]
  the original list:  [[Brook], [Nami]]
Deep copying
  same list object?   false
  same first person?  false
  the copied list:    [[Brook], [Nami], [Franky]]
  the original list:  [[Usopp], [Nami]]
```

- **Aliasing.** `experiment(..., people, people)` passes the same list twice. `copy == people` is `true`: there is one list and two names for it, so both the rename and the added Franky appear "in both lists".
- **Shallow copying.** `copyList` makes a new `ArrayList` from `people`. (The following `Collections.copy(result, people)` copies the element references once more into the new list; it is harmless but redundant, because the constructor already did that — worth noticing when you read older code.) Now `copy == people` is `false`: there are two lists, and adding Franky to one does not affect the other. But `copy.get(0) == people.get(0)` is `true`: both lists refer to the *same* Usopp object, so renaming him through the copy renames him in the original too. The original list becomes `[Brook, Nami]`.
- **Deep copying.** `deepCopy` also makes a new list, and then replaces each slot with `new Person(people.get(i).getName())` — a new object with the same data. Now nothing is shared, and the original stays `[Usopp, Nami]`.

Neither kind of copy is "the right one". A shallow copy is cheap and correct whenever the elements are **immutable** (strings, numbers, the value objects of Section 13.9): sharing an object

that cannot change is perfectly safe. A deep copy is needed when the elements are mutable and the copy must be independent. Notice how immutability simplifies the question: if `Person` could not be renamed, the shallow copy and the deep copy would be indistinguishable. That observation is our bridge to Section 13.8.

Java has no general-purpose deep copy. Each class must decide what copying it means — a *copy constructor* such as `new Person(other)` is the usual tool — because only the class knows which of its fields are values and which are references to things it shares on purpose [9]. Python does have one, in the standard module `copy`, as **Listing 13.14** shows.

Listing 13.14 `copy_demo.py`: aliasing, `list(...)` as a shallow copy, `copy.deepcopy`, and the same three levels for a nested list.

```
"""copy_demo.py: aliasing, shallow copies and deep copies."""
import copy


class Person:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f"[{self.name}]"


def crew():
    return [Person("Usopp"), Person("Nami")]


def experiment(title, people, copied):
    print(title)
    print("  same list object?  ", copied is people)
    print("  same first person? ", copied[0] is people[0])
    copied[0].name = "Brook"             # Update the copied list: Usopp -> Brook
    copied.append(Person("Franky"))      # and add someone to the copy only
    print("  the copied list:   ", copied)
    print("  the original list: ", people)


def main():
    people = crew()
    experiment("Aliasing (no copy at all)", people, people)
    people = crew()
    experiment("Shallow copying", people, list(people))      # or copy.copy(people)
    people = crew()
    experiment("Deep copying", people, copy.deepcopy(people))

    # Nested lists show the same three levels.
    grid = [[1, 2], [3, 4]]
    shallow, deep = copy.copy(grid), copy.deepcopy(grid)
    grid[0][0] = 99
    print("grid:", grid, " shallow:", shallow, " deep:", deep)


if __name__ == "__main__":
    main()
```

```
Aliasing (no copy at all)
  same list object?   True
  same first person?  True
  the copied list:    [[Brook], [Nami], [Franky]]
  the original list:  [[Brook], [Nami], [Franky]]
Shallow copying
  same list object?   False
  same first person?  True
  the copied list:    [[Brook], [Nami], [Franky]]
  the original list:  [[Brook], [Nami]]
Deep copying
  same list object?   False
  same first person?  False
  the copied list:    [[Brook], [Nami], [Franky]]
  the original list:  [[Usopp], [Nami]]
grid: [[99, 2], [3, 4]]  shallow: [[99, 2], [3, 4]]  deep: [[1, 2], [3, 4]]
```

`list(people)` (or `people.copy()`, `people[:]` and `copy.copy(people)`, which all mean the same thing for a list) makes a shallow copy; `copy.deepcopy(people)` walks the whole structure and copies every object it finds, keeping track of objects it has already copied so that shared references and cycles are copied correctly [4]. The last line shows the same three levels with a list of lists, a structure beginners often build for grids and tables: changing `grid[0][0]` shows through the shallow copy, because both outer lists share the inner lists, while the deep copy is unaffected.

13.8 Immutability from Zero

Though we may not have noticed it, we have been using immutable objects since Chapter 3.

> **DEFINITION.** An object is **immutable** if its state cannot change after it has been created. An object whose state can change is **mutable**. (Remember that an *object* is different from a *variable*: a variable that refers to an immutable object can still be made to refer to a different object.)

`String` is the everyday example. Once a string is created it cannot be modified; every method that seems to modify it — `toUpperCase()`, `trim()`, `replace(...)` — returns a *new* string and leaves the original untouched. We can still create new strings whenever we need them, and a variable such as `day` in `day = day.toUpperCase()` can be made to refer to the new one; but the string `"monday"` itself stays `"monday"` forever. The same is true of `Integer`, `Long`, `Double`, `LocalDate` and the other date-time classes in Java, and of `int`, `float`, `str`, `tuple` and `frozenset` in Python. Mutable objects, on the other hand — `ArrayList`, `HashMap`, `StringBuilder`, Python's `list`, `dict` and `set`, and every class we have written with a setter — can have their state altered after they are created.

13.8.1 Why would anyone want an object that cannot change?

The Java documentation puts it strongly: "Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code" [11]. Why would that be? What possible benefits can arise from *not* being allowed to modify an object? There are four, and each deserves a sentence of its own.

1. Immutable objects make good keys. Chapter 9 warned that an object must never change while it is in use as a key in a map or as an element of a set. An immutable object *cannot* change, so the warning takes care of itself. Section 13.9 shows exactly what goes wrong when the rule is broken.

2. We do not need to worry about what happens when we pass an object to a method. Suppose we pass a list to a method called `medianAverage`. To find the median — the middle value — the method has to sort the values. It may sort *our* list, or it may make its own copy and sort that. The first behavior is undesirable, because the order of our list might hold significance. With strings this is never a problem: the method can create new strings based on the string parameter, but it cannot alter the string we gave it. **Listing 13.15** shows the problem with a list of kilograms washed per day, in which the order *is* the information.

Listing 13.15 `MedianDemo.java`: a median method that sorts its caller's list, one that sorts a copy, and an unmodifiable view that turns the hidden side effect into an exception.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MedianDemo {
    // Sorts the caller's list to find the middle: a hidden side effect.
    static double medianAverageCareless(List<Integer> values) {
        Collections.sort(values);
        return middle(values);
    }

    // Sorts its own copy: the caller's list is left exactly as it was.
    static double medianAverage(List<Integer> values) {
        List<Integer> sorted = new ArrayList<>(values);
        Collections.sort(sorted);
        return middle(sorted);
    }

    private static double middle(List<Integer> sorted) {
        int n = sorted.size();
        if (n == 0) {
            throw new IllegalArgumentException("no values");
        }
        return n % 2 == 1 ? sorted.get(n / 2)
                          : (sorted.get(n / 2 - 1) + sorted.get(n / 2)) / 2.0;
    }

    public static void main(String[] args) {
        // Kilograms washed per day, Monday to Saturday: the ORDER is the information.
        List<Integer> kgPerDay = new ArrayList<>(List.of(42, 35, 51, 38, 60, 47));
        System.out.println("week:    " + kgPerDay);
        System.out.println("median:  " + medianAverageCareless(kgPerDay));
        System.out.println("week:    " + kgPerDay + "   <- which day was Monday?");

        kgPerDay = new ArrayList<>(List.of(42, 35, 51, 38, 60, 47));
        System.out.println("median:  " + medianAverage(kgPerDay));
        System.out.println("week:    " + kgPerDay + "   <- untouched");
    }
```

Listing 13.15 (Continued) *MedianDemo.java: a median method that sorts its caller's list, one that sorts a copy, and an unmodifiable view that turns the hidden side effect into an exception.*

```
        // An unmodifiable view turns the hidden side effect into a loud error.
        List<Integer> guarded = Collections.unmodifiableList(kgPerDay);
        try {
            medianAverageCareless(guarded);
        } catch (UnsupportedOperationException e) {
            System.out.println("careless median on a guarded list: "
                    + e.getClass().getSimpleName());
        }

        // Strings are immutable: a method can only build NEW strings from them.
        String day = "monday";
        String shout = day.toUpperCase();
        System.out.println(day + " " + shout);
    }
}
```

```
week:    [42, 35, 51, 38, 60, 47]
median:  44.5
week:    [35, 38, 42, 47, 51, 60]   <- which day was Monday?
median:  44.5
week:    [42, 35, 51, 38, 60, 47]   <- untouched
careless median on a guarded list: UnsupportedOperationException
monday MONDAY
```

`medianAverageCareless` returns the right median, 44.5 (the average of the middle values 42 and 47), but afterwards the week is sorted and nobody can tell which day was Monday. Such a **side effect** on an argument is hard to find, because the method that broke the list returned the correct answer. `medianAverage` sorts its own copy and leaves the caller's list alone. And the last block shows the defensive move from the caller's side: pass an *unmodifiable view* (Section 13.8.2), and a careless method that tries to sort it fails loudly with `UnsupportedOperationException` instead of silently corrupting the data. **Listing 13.16** tells the same story in Python.

Listing 13.16 `median_demo.py`: `list.sort()` changes the caller's list, `sorted()` does not, and a tuple cannot be sorted in place at all.

```
"""median_demo.py: a function that changes its argument, and one that does not."""


def median_average_careless(values):
    values.sort()                       # sorts the caller's list: a hidden side effect
    return _middle(values)


def median_average(values):
    return _middle(sorted(values))      # sorted() builds a new list


def _middle(ordered):
    n = len(ordered)
    if n == 0:
        raise ValueError("no values")
    if n % 2 == 1:
        return ordered[n // 2]
    return (ordered[n // 2 - 1] + ordered[n // 2]) / 2


def main():
    # Kilograms washed per day, Monday to Saturday: the ORDER is the information.
    kg_per_day = [42, 35, 51, 38, 60, 47]
    print("week:   ", kg_per_day)
    print("median: ", median_average_careless(kg_per_day))
    print("week:   ", kg_per_day, "  <- which day was Monday?")

    kg_per_day = [42, 35, 51, 38, 60, 47]
    print("median: ", median_average(kg_per_day))
    print("week:   ", kg_per_day, "  <- untouched")

    # A tuple cannot be sorted in place, so the careless version fails loudly.
    guarded = tuple(kg_per_day)
    try:
        median_average_careless(guarded)
    except AttributeError as e:
        print("careless median on a tuple:", e)
    print("median of the tuple:", median_average(guarded))

    day = "monday"
    shout = day.upper()
    print(day, shout)
```

Listing 13.16 (Continued) *median_demo.py: list.sort() changes the caller's list, sorted() does not, and a tuple cannot be sorted in place at all.*

```
if __name__ == "__main__":
    main()
```

```
week:    [42, 35, 51, 38, 60, 47]
median:  44.5
week:    [35, 38, 42, 47, 51, 60]   <- which day was Monday?
median:  44.5
week:    [42, 35, 51, 38, 60, 47]   <- untouched
careless median on a tuple: 'tuple' object has no attribute 'sort'
median of the tuple: 44.5
monday MONDAY
```

Python makes the distinction visible in the names: the *method* `values.sort()` sorts in place and returns `None`, while the *function* `sorted(values)` builds and returns a new list. A tuple has no `sort` method at all, so passing a tuple turns the careless version into an immediate `AttributeError`.

3. Immutable objects can be shared between threads. Chapter 14 introduces threads: pieces of code that run at the same time as the main program. If two threads can change the same object, their changes can interleave and corrupt it. A simple analogy is a shared spreadsheet of results that two clerks open at the same time. Clerk A finishes first and saves; clerk B finishes a minute later and saves his copy, overwriting A's changes, and the sheet is left in a broken state. An immutable object is like a printed, signed document: any number of people can read it at the same time, and nobody can scribble on it. Sharing an immutable object between threads needs no locks at all [12].

4. Immutable objects are simple. An immutable object has exactly one state, the one it was born with. If its constructor checks that the state is valid, it is valid forever; there is no sequence of method calls that can break it, so there are no such sequences to test. Chapter 7 called a promise of this kind an *invariant*; an immutable object keeps its invariants for free [9].

Programmers sometimes worry that creating new objects instead of changing old ones is costly. There is some overhead, but modern virtual machines create and dispose of small, short-lived objects very efficiently, and the cost is repaid by the copies we no longer make: an immutable object never needs a defensive copy.

Mutability is still the right choice for some things. A **bank account** exists precisely to change its balance over time; a **washing machine** has a door that opens and a load that grows. Such objects have *identity* — this machine, not an equal one — and a life story. The distinction between objects defined by their identity and objects defined by their value is the theme of Section 13.9, and Exercise 13.7 asks you to classify several classes.

13.8.2 final is not immutable; views and copies

Java's first tool for immutability is the keyword `final`. On a field or a variable, `final` tells the compiler that its value may be assigned only once — for a field, in its declaration or in a constructor — and never changed afterwards. That is the most robust way to enforce immutability in a class; the simplest is to provide no setter methods, and a careful class does both.

But `final` fixes the *variable*, not the *object it refers to*. A `final List<String>` always refers to the same list, and that list can still grow and shrink. For collections, Java therefore offers two separate tools that are easily confused, and Python has counterparts to both. **Listing 13.17** lays them out.

Listing 13.17 `ImmutableCollections.java`: `final` versus immutable, an unmodifiable view that shows later changes, an unmodifiable copy that does not, the `of` factory methods, and shallow unmodifiability.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class ImmutableCollections {
    static void attempt(String what, Runnable action) {
        try {
            action.run();
            System.out.println(what + ": allowed");
        } catch (RuntimeException e) {
            System.out.println(what + ": " + e.getClass().getSimpleName());
        }
    }

    public static void main(String[] args) {
        // 1. final fixes the VARIABLE, not the object it refers to.
        final List<String> services = new ArrayList<>(List.of("WF", "IR"));
        services.add("DC");                      // allowed: the list itself is mutable
        // services = new ArrayList<>();         // would not compile: services is final
        System.out.println("final list after add: " + services);

        // 2. An unmodifiable VIEW: no changes through it, but it shows changes behind it.
        List<String> view = Collections.unmodifiableList(services);
        attempt("view.add(\"WDI\")", () -> view.add("WDI"));
        services.add("WDI");                     // change the list behind the view
        System.out.println("view now shows: " + view);

        // 3. An unmodifiable COPY: a snapshot that nothing can change.
        List<String> snapshot = List.copyOf(services);
        services.remove("IR");
        System.out.println("services: " + services + "  snapshot: " + snapshot);
        attempt("snapshot.set(0, \"XX\")", () -> snapshot.set(0, "XX"));

        // 4. The factory methods of, and their rules.
        List<Integer> kg = List.of(3, 2, 3);
        Set<String> codes = Set.of("WF", "IR", "DC");
        Map<String, Integer> rate = Map.of("WF", 7000, "IR", 5000);
        System.out.println(kg + " " + codes.size() + " codes, WF costs " + rate.get("WF"));
        attempt("List.of with a null", () -> List.of("WF", null));
        attempt("Set.of with a duplicate", () -> Set.of("WF", "WF"));
        attempt("rate.put(\"DC\", 15000)", () -> rate.put("DC", 15000));
```

Listing 13.17 (Continued) *ImmutableCollections.java: final versus immutable, an unmodifiable view that shows later changes, an unmodifiable copy that does not, the of factory methods, and shallow unmodifiability.*

```
        // 5. Unmodifiable is shallow: the elements may still be mutable.
        List<StringBuilder> notes = List.of(new StringBuilder("rush"));
        notes.get(0).append(" order");
        System.out.println("inside an unmodifiable list: " + notes);
    }
}
```

```
final list after add: [WF, IR, DC]
view.add("WDI"): UnsupportedOperationException
view now shows: [WF, IR, DC, WDI]
services: [WF, DC, WDI]  snapshot: [WF, IR, DC, WDI]
snapshot.set(0, "XX"): UnsupportedOperationException
[3, 2, 3] 3 codes, WF costs 7000
List.of with a null: NullPointerException
Set.of with a duplicate: IllegalArgumentException
rate.put("DC", 15000): UnsupportedOperationException
inside an unmodifiable list: [rush order]
```

- **Block 1** adds `"DC"` to a `final` list without complaint. Only the commented-out line, which would make `services` refer to a different list, is forbidden.
- **Block 2** wraps the list in `Collections.unmodifiableList(services)`. The result is a **view**: a thin object that forwards reading methods to the real list and throws `UnsupportedOperationException` from every changing method (`add`, `remove`, `set`, `clear`, `sort`, ...). A view is *read-only*, not *immutable*: when the real list behind it changes, the view shows the change, as `[WF, IR, DC, WDI]` demonstrates. A view is the right tool when a class wants to let others **see** its live data without letting them **touch** it.
- **Block 3** uses `List.copyOf(services)`, which makes an **unmodifiable copy**: a snapshot that neither the caller nor anyone else can change. Removing `IR` from `services` afterwards does not affect `snapshot`. A copy is the right tool when a class must be sure that data it received will never change behind its back.
- **Block 4** shows the factory methods `List.of`, `Set.of` and `Map.of`, added in Java 9, which create small unmodifiable collections directly [13]. They are strict: they reject `null` elements (`NullPointerException`), `Set.of` rejects duplicates (`IllegalArgumentException`), and every changing method throws `UnsupportedOperationException`.
- **Block 5** is a reminder that all of these are **shallow**: an unmodifiable list of `StringBuilder`s cannot gain or lose elements, but each `StringBuilder` in it can still be changed. A collection is only as immutable as its elements.

Listing 13.18 shows the same ideas in Python.

Listing 13.18 `immutable_collections.py`: `Final` as a hint, `MappingProxyType` as a read-only view, `tuple` and `frozenset` as immutable copies, hashability, and shallow immutability.

```
"""immutable_collections.py: tuples, frozensets and read-only views."""
from types import MappingProxyType
from typing import Final


def attempt(what, action):
    try:
        action()
        print(f"{what}: allowed")
    except (TypeError, AttributeError) as e:
        print(f"{what}: {type(e).__name__}: {e}")


def main():
    # 1. Final is only a promise to the type checker; Python itself does not enforce it.
    SERVICES: Final = ["WF", "IR"]
    SERVICES.append("DC")                  # the list is mutable anyway
    print("Final list after append:", SERVICES)

    # 2. A read-only VIEW of a dict: no changes through it, but it shows changes behind it.
    rates = {"WF": 7000, "IR": 5000}
    view = MappingProxyType(rates)
    def change_view():
        view["DC"] = 15000
    attempt('view["DC"] = 15000', change_view)
    rates["DC"] = 15000
    print("view now shows:", dict(view))

    # 3. Immutable COPIES: tuple and frozenset.
    snapshot = tuple(SERVICES)
    SERVICES.remove("IR")
    print("services:", SERVICES, " snapshot:", snapshot)
    def change_snapshot():
        snapshot[0] = "XX"
    attempt("snapshot[0] = 'XX'", change_snapshot)
    codes = frozenset({"WF", "IR", "DC"})
    attempt("codes.add('WDI')", lambda: codes.add("WDI"))
    print("frozenset keeps one of each:", sorted(frozenset(["WF", "WF", "IR"])))

    # 4. Immutable things can be dictionary keys; mutable ones cannot.
    price_by_pair = {("WF", "express"): 10500, frozenset({"wash", "iron"}): 10000}
    print("tuple key:", price_by_pair[("WF", "express")])
    attempt("list as a key", lambda: {["WF", "express"]: 10500})
```

Listing 13.18 (Continued) *immutable_collections.py: Final as a hint, MappingProxyType as a read-only view, tuple and frozenset as immutable copies, hashability, and shallow immutability.*

```
    # 5. A tuple is shallow: the elements may still be mutable.
    notes = (["rush"],)
    notes[0].append("order")
    print("inside a tuple:", notes)
    attempt("hash(notes)", lambda: hash(notes))


if __name__ == "__main__":
    main()
```

```
Final list after append: ['WF', 'IR', 'DC']
view["DC"] = 15000: TypeError: 'mappingproxy' object does not support item assignment
view now shows: {'WF': 7000, 'IR': 5000, 'DC': 15000}
services: ['WF', 'DC']  snapshot: ('WF', 'IR', 'DC')
snapshot[0] = 'XX': TypeError: 'tuple' object does not support item assignment
codes.add('WDI'): AttributeError: 'frozenset' object has no attribute 'add'
frozenset keeps one of each: ['IR', 'WF']
tuple key: 10500
list as a key: TypeError: unhashable type: 'list'
inside a tuple: (['rush', 'order'],)
hash(notes): TypeError: unhashable type: 'list'
```

Python has no `final` keyword; the annotation `typing.Final` is checked only by tools such as mypy, never at run time, and would fix only the name anyway. `types.MappingProxyType(rates)` is Python's read-only **view** of a dictionary: assignment through it raises `TypeError`, yet it shows the later change to `rates`. The immutable **copies** are the built-in types `tuple` (an immutable sequence) and `frozenset` (an immutable set), and there is no `frozendict` in the standard library. Because tuples and frozensets cannot change, they can be dictionary keys, as `price_by_pair` shows; a `list` cannot, and the attempt raises `TypeError: unhashable type: 'list'`. Block 5 shows Python's version of shallowness: a tuple containing a list cannot change *which* list it holds, but the list can change, and such a tuple is therefore not hashable either. **Table 13.3** summarizes the tools.

Table 13.3 Tools for immutability in Java and Python.

Need	Java	Python
A name that is assigned once	`final` variable or field	convention (`UPPER_CASE`), `typing.Final` for checkers
Read-only live view of a collection	`Collections.unmodifiableList/Set/Map(c)`	`types.MappingProxyType(d)` for dictionaries
Immutable snapshot of a collection	`List.copyOf`, `Set.copyOf`, `Map.copyOf`	`tuple(x)`, `frozenset(x)`
Small immutable collection literal	`List.of`, `Set.of`, `Map.of`	`(a, b, c)`, `frozenset({a, b})`
Immutable class with little code	`record` (Java 16+)	`@dataclass(frozen=True)`, `typing.NamedTuple`
Immutable class written by hand	`final` class, `private final` fields, no setters	read-only `@property` or `__slots__` plus no setters

13.8.3 Writing an immutable class

To make a class immutable, follow five rules [9]:

1. **Provide no methods that change the object's state** — no setters, no `add`, no `increment`.
2. **Make sure the class cannot be extended**, usually by declaring it `final`; otherwise a subclass could add mutable state and override methods so that "your" immutable object changes after all.
3. **Make all fields `private` and `final`**, so the compiler itself guarantees that no code, not even inside the class, reassigns them after construction.
4. **Guard every mutable component.** If a field refers to a mutable object such as a list, make a **defensive copy** when you receive it in the constructor and never hand out the original; return an unmodifiable view or a copy instead.

5. **Build the object completely in the constructor**, and do not let `this` escape before the constructor finishes.

For an object to be fully immutable, its fields must be primitives or references to immutable objects (rule 4 takes care of the rest). **Listing 13.19** applies the rules to a hand-written class, then shows how Java *records* do most of the work for us.

Listing 13.19 `ImmutableClasses.java`: an immutable `Point` written by hand, a `Student` with a life-long id, a class that leaks its list, and a record with a defensive copy in its compact constructor.

```
import java.util.ArrayList;
import java.util.List;

public class ImmutableClasses {
    // An immutable class written by hand: the five rules in action.
    static final class Point {                  // rule 2: final class, no subclasses
        private final int x;                    // rule 3: private final fields
        private final int y;

        Point(int x, int y) {
            this.x = x;
            this.y = y;
        }

        int getX() {                            // rule 1: getters, no setters
            return x;
        }

        int getY() {
            return y;
        }

        Point withX(int newX) {                 // "changing" means making a new object
            return new Point(newX, y);
        }

        @Override
        public String toString() {
            return "(" + x + ", " + y + ")";
        }
    }

    // Every student gets the next id once, at birth, and keeps it for life.
    static final class Student {
        private static int ID_COUNTER = 1;
        private final int id = ID_COUNTER++;
        private final String name;

        Student(String name) {
            this.name = name;
        }

        int getId() {
            return id;
        }
```

Listing 13.19 (Continued) *ImmutableClasses.java: an immutable Point written by hand, a Student with a life-long id, a class that leaks its list, and a record with a defensive copy in its compact constructor.*

```
        String getName() {
            return name;
        }
    }

    // Looks immutable, but leaks its list in both directions.
    static final class LeakyCustomer {
        private final String name;
        private final List<String> phones;

        LeakyCustomer(String name, List<String> phones) {
            this.name = name;
            this.phones = phones;               // keeps the caller's list
        }

        List<String> getPhones() {
            return phones;                      // hands out its own list
        }
    }

    // A record with a compact constructor that makes a defensive copy.
    record Customer(String name, List<String> phones) {
        Customer {                              // rule 4: copy mutable input
            if (name == null || name.isBlank()) {
                throw new IllegalArgumentException("name required");
            }
            phones = List.copyOf(phones);       // unmodifiable snapshot
        }
    }

    public static void main(String[] args) {
        Point p = new Point(1, 2);
        Point q = p.withX(5);
        System.out.println("p = " + p + ", q = " + q);

        Student a = new Student("Robin");
        Student b = new Student("Franky");
        System.out.println(a.getName() + " has id " + a.getId()
                + ", " + b.getName() + " has id " + b.getId());

        List<String> phones = new ArrayList<>(List.of("0812-1111"));
        LeakyCustomer leaky = new LeakyCustomer("Siti", phones);
        phones.add("0813-2222");                // the caller changes its own list...
        leaky.getPhones().add("0000-HACK");     // ...and anyone can change the getter's
        System.out.println("leaky phones: " + leaky.getPhones());
```

Listing 13.19 (Continued) *ImmutableClasses.java: an immutable Point written by hand, a Student with a life-long id, a class that leaks its list, and a record with a defensive copy in its compact constructor.*

```
        phones = new ArrayList<>(List.of("0812-1111"));
        Customer safe = new Customer("Siti", phones);
        phones.add("0813-2222");
        System.out.println("safe phones:  " + safe.phones());
        try {
            safe.phones().add("0000-HACK");
        } catch (UnsupportedOperationException e) {
            System.out.println("safe.phones().add: " + e.getClass().getSimpleName());
        }

        // Records give equals, hashCode and toString for free, all from the fields.
        Customer twin = new Customer("Siti", List.of("0812-1111"));
        System.out.println(safe + " equals twin? " + safe.equals(twin));
        try {
            new Customer(" ", List.of());
        } catch (IllegalArgumentException e) {
            System.out.println("blank name: " + e.getMessage());
        }
    }
}
```

```
p = (1, 2), q = (5, 2)
Robin has id 1, Franky has id 2
leaky phones: [0812-1111, 0813-2222, 0000-HACK]
safe phones:  [0812-1111]
safe.phones().add: UnsupportedOperationException
Customer[name=Siti, phones=[0812-1111]] equals twin? true
blank name: name required
```

`Point` follows rules 1–3 and 5 visibly, and needs nothing for rule 4, because `int`s are values. To "change" a point we call `withX(5)`, which returns a *new* `Point`; `p` itself is still `(1, 2)`. The `withX` style — a method that returns a modified copy — is how every immutable type offers change: `String.toUpperCase`, `LocalDate.plusDays`, `BigDecimal.add` all work this way.

`Student` shows a pattern for identifiers. `ID_COUNTER` is a `static` field shared by the class and incremented once per student; the instance field `private final int id = ID_COUNTER++;` takes the next number when the student is created and keeps it for life. The `name` is a `String`, which is immutable, so the whole `Student` is immutable.

`LeakyCustomer` is the classic trap. Its fields are `private final`, it has no setters, and it *looks* immutable. But its constructor stores the caller's list, and its getter hands out that same list. The output shows the result: the caller added `0813-2222` to its own list *after* constructing the customer, and a stranger added `0000-HACK` through the getter, and both changes landed inside the "immutable" object. `final` guarded the *reference*, not the list.

`Customer` is a **record**, introduced in Java 16 [14]. The line `record Customer(String name, List<String> phones) { ... }` declares a class that is automatically `final`, with `private final` fields `name` and `phones`, a constructor that sets them, accessor methods `name()` and `phones()`, and `equals`, `hashCode` and `toString` computed from the fields — everything Chapter 7 wrote by hand for value classes. The block `Customer { ... }` is a **compact constructor**: it runs before the fields are assigned, and it may validate and replace the parameters. Ours rejects a blank name and replaces `phones` with `List.copyOf(phones)`, which is rule 4 in one line. Now the caller's later change does not reach the record, and `safe.phones().add(...)` throws. Records are Java's preferred way to write small immutable data classes; the only thing they cannot do for you is the defensive copy, because the compiler cannot know which of your fields are mutable.

Listing 13.20 writes the same classes in Python.

Listing 13.20 `immutable_classes.py`: frozen dataclasses with `replace`, an id fixed at birth, a leaky class, a defensive copy in `__post_init__`, and `NamedTuple`.

```
"""immutable_classes.py: frozen dataclasses, defensive copies and NamedTuple."""
from dataclasses import FrozenInstanceError, dataclass, field, replace
from itertools import count
from typing import NamedTuple


@dataclass(frozen=True)
class Point:
    x: int
    y: int

    def with_x(self, new_x):
        return replace(self, x=new_x)    # a new Point; self is untouched


_ids = count(1)


@dataclass(frozen=True)
class Student:
    name: str
    id: int = field(default_factory=lambda: next(_ids))   # fixed at birth


class LeakyCustomer:
    """Looks immutable, but leaks its list in both directions."""

    def __init__(self, name, phones):
        self._name = name
        self._phones = phones            # keeps the caller's list

    @property
    def phones(self):
        return self._phones              # hands out its own list


@dataclass(frozen=True)
class Customer:
    name: str
    phones: tuple
```

Listing 13.20 (Continued) *immutable_classes.py: frozen dataclasses with replace, an id fixed at birth, a leaky class, a defensive copy in __post_init__, and NamedTuple.*

```
    def __post_init__(self):
        if not self.name or not self.name.strip():
            raise ValueError("name required")
        # frozen blocks normal assignment, so use object.__setattr__ once, here.
        object.__setattr__(self, "phones", tuple(self.phones))


class Money(NamedTuple):                # a tuple with named fields: immutable too
    rupiah: int


def main():
    p = Point(1, 2)
    q = p.with_x(5)
    print(f"p = {p}, q = {q}")
    try:
        p.x = 9
    except FrozenInstanceError as e:
        print("p.x = 9:", type(e).__name__, "-", e)

    a, b = Student("Robin"), Student("Franky")
    print(f"{a.name} has id {a.id}, {b.name} has id {b.id}")

    phones = ["0812-1111"]
    leaky = LeakyCustomer("Siti", phones)
    phones.append("0813-2222")
    leaky.phones.append("0000-HACK")
    print("leaky phones:", leaky.phones)

    phones = ["0812-1111"]
    safe = Customer("Siti", phones)
    phones.append("0813-2222")
    print("safe phones: ", safe.phones)
    try:
        safe.phones.append("0000-HACK")
    except AttributeError as e:
        print("safe.phones.append:", type(e).__name__)

    twin = Customer("Siti", ["0812-1111"])
    print(safe, "equals twin?", safe == twin, " same hash?", hash(safe) == hash(twin))
    try:
        Customer(" ", [])
    except ValueError as e:
        print("blank name:", e)
```

Listing 13.20 (Continued) *immutable_classes.py: frozen dataclasses with replace, an id fixed at birth, a leaky class, a defensive copy in __post_init__, and NamedTuple.*

```
    m = Money(7000)
    print(m, m.rupiah, m._replace(rupiah=5000), m)


if __name__ == "__main__":
    main()
```

```
p = Point(x=1, y=2), q = Point(x=5, y=2)
p.x = 9: FrozenInstanceError - cannot assign to field 'x'
Robin has id 1, Franky has id 2
leaky phones: ['0812-1111', '0813-2222', '0000-HACK']
safe phones:  ('0812-1111',)
safe.phones.append: AttributeError
Customer(name='Siti', phones=('0812-1111',)) equals twin? True  same hash? True
blank name: name required
Money(rupiah=7000) 7000 Money(rupiah=5000) Money(rupiah=7000)
```

`@dataclass(frozen=True)` is Python's counterpart of a record [15]: it generates `__init__`, `__repr__`, `__eq__` and `__hash__` from the annotated fields, and `frozen=True` makes every assignment to a field raise `FrozenInstanceError`, as `p.x = 9` shows. `dataclasses.replace(self, x=new_x)` builds a modified copy, the `withX` pattern. `Student` gets its id from `field(default_factory=lambda: next(_ids))`, where `_ids = count(1)` is an infinite iterator from `itertools` that yields 1, 2, 3, … — the counter of the Java version, written as an iterator.

The defensive copy lives in `__post_init__`, a method that dataclasses call right after `__init__`. Because the object is already frozen at that point, the ordinary `self.phones = ...` would fail; the documented way out is `object.__setattr__(self, "phones", tuple(self.phones))`, used once, inside the class, during construction. Converting the list to a tuple both copies it and makes it immutable, so `safe.phones.append` fails with `AttributeError`. Finally, `typing.NamedTuple` creates a tuple subclass with named fields; it is immutable because tuples are, and `_replace` makes modified copies.

> ⚠ **WARNING.** Python's immutability is a strong convention, not an iron wall: `object.__setattr__` works on a frozen dataclass from anywhere, not just from `__post_init__`. Frozen dataclasses protect against *mistakes*, which is what matters in practice; they do not protect against deliberate tampering, and Java's `final` fields do not fully do so either (reflection can reach them). Design for honest programmers who make mistakes.

13.9 Value Objects, Equality and Hashing

Some objects are defined by **who they are**: two washing machines of the same model with the same settings are still two machines, and it matters which one is broken. Others are defined entirely by **what they are**: two amounts of Rp7,000 are the same amount; two dates of 21 September 2026 are the same date; it makes no sense to ask *which* Rp7,000 you have.

> **DEFINITION.** A **value object** is an object whose identity does not matter: it is completely described by the values of its fields, two value objects with equal fields are interchangeable, and it is (almost always) immutable. An **entity**, by contrast, has an identity that persists while its fields change [16].

Money, prices, dates, weights, coordinates and colors are value objects; customers, machines and bank accounts are entities. Value objects should be immutable: if you could change "the" Rp7,000, every place that shared it would change with it.

Because a value object is described by its fields, its `equals` (Python `__eq__`) must compare the fields, and its `hashCode` (`__hash__`) must be computed from the same fields. Chapter 7 stated the rule; now we can see why it exists and what goes wrong with mutable keys. The rule itself is one implication:

$$a.\text{equals}(b) \;\Rightarrow\; h(a) = h(b) \tag{13.6}$$

where h is the hash function. Equation (13.6) says that equal objects must have equal hash codes. (The converse need not hold: different objects may share a hash code.) The reason is how a hash table works [17]. A `HashMap` or a Python `dict` keeps an array of m **buckets**. To store or find a key k, it computes the key's hash code and uses it to choose one bucket:

$$i = h(k) \bmod m \tag{13.7}$$

and then searches *only* bucket i, comparing keys with `equals`. (Java's `HashMap` first mixes the high bits of $h(k)$ into the low ones and uses a power of two for m, so that the remainder is computed with a bit mask; for the small hash codes below, the result is exactly Equation (13.7).) This is why hash tables are fast — one bucket instead of the whole table — and why Equation (13.6) is essential: if two equal keys could have different hash codes, Equation (13.7) would send them to different buckets, and the table could never find one when given the other.

Now suppose a key **changes after it has been stored**. Its hash code is computed from its fields, so the hash code changes too. The entry is still sitting in the bucket chosen by the *old* hash code, but every lookup goes to the bucket chosen by the *new* one. **Listing 13.21** does exactly this.

Listing 13.21 `MutableKey.java`: a mutable `Spot` used as a key in a `HashMap` and a `HashSet`, changed while stored; then an immutable record used correctly.

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Objects;
import java.util.Set;

public class MutableKey {
    // A mutable point with equals and hashCode computed from its fields.
    static final class Spot {
        int row, col;

        Spot(int row, int col) {
            this.row = row;
            this.col = col;
        }

        @Override
        public boolean equals(Object o) {
            return o instanceof Spot s && s.row == row && s.col == col;
        }

        @Override
        public int hashCode() {
            return Objects.hash(row, col);
        }

        @Override
        public String toString() {
            return "Spot(" + row + "," + col + ")";
        }
    }

    // The immutable version: a record's fields are final.
    record Shelf(int row, int col) { }

    public static void main(String[] args) {
        Spot spot = new Spot(1, 2);
        Map<Spot, String> rack = new HashMap<>();
        rack.put(spot, "Siti's bag");
        Set<Spot> used = new HashSet<>();
        used.add(spot);
        System.out.println("hashCode before: " + spot.hashCode());
```

Listing 13.21 (Continued) *MutableKey.java: a mutable Spot used as a key in a HashMap and a HashSet, changed while stored; then an immutable record used correctly.*

```
        spot.row = 3;                           // mutate the key while it is in the map
        System.out.println("hashCode after:  " + spot.hashCode());
        System.out.println("rack.get(spot):              " + rack.get(spot));
        System.out.println("rack.get(new Spot(1, 2)):    " + rack.get(new Spot(1, 2)));
        System.out.println("rack.containsKey(spot):      " + rack.containsKey(spot));
        System.out.println("but the entry is still there: " + rack);
        used.add(spot);                         // the "same" key goes in twice
        System.out.println("used = " + used + ", size " + used.size());

        Map<Shelf, String> shelves = new HashMap<>();
        Shelf shelf = new Shelf(1, 2);
        shelves.put(shelf, "Siti's bag");
        // shelf.row = 3;                       // would not compile: records are immutable
        Shelf moved = new Shelf(3, 2);          // "moving" means a new key
        shelves.put(moved, shelves.remove(shelf));
        System.out.println("shelves = " + shelves);
        System.out.println("equal records, equal hashes: "
                + (new Shelf(3, 2).hashCode() == moved.hashCode()));
    }
}
```

```
hashCode before: 994
hashCode after:  1056
rack.get(spot):              null
rack.get(new Spot(1, 2)):    null
rack.containsKey(spot):      false
but the entry is still there: {Spot(3,2)=Siti's bag}
used = [Spot(3,2), Spot(3,2)], size 2
shelves = {Shelf[row=3, col=2]=Siti's bag}
equal records, equal hashes: true
```

`Spot` is a rack position — row and column — with `equals` and `hashCode` correctly computed from both fields (`Objects.hash(row, col)` is $31 \cdot (31 \cdot 1 + \text{row}) + \text{col}$). We store Siti's bag under `Spot(1, 2)`, whose hash code is $31 \cdot 32 + 2 = 994$. Then `spot.row = 3` changes the key in place, and its hash code becomes $31 \cdot 34 + 2 = 1056$. **Figure 13.4** shows where everything is in a map with the default 16 buckets.

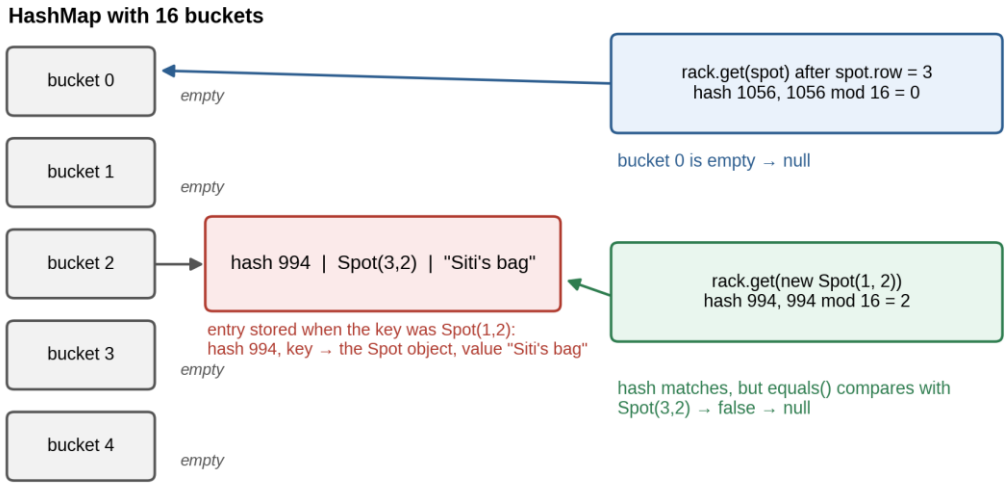


Figure 13.4 A mutated key is lost. The entry was filed in bucket 994 mod 16 = 2; the changed key's hash sends lookups to bucket 1056 mod 16 = 0, which is empty, while a lookup with the old value finds the right bucket but fails the `equals` test.

- `rack.get(spot)` computes the *new* hash, 1056, looks in bucket 1056 mod 16 = 0, finds nothing, and returns `null`.
- `rack.get(new Spot(1, 2))` computes the *old* hash, 994, and looks in the right bucket, 2. `HashMap` stores each entry's hash code alongside it, so the hashes match; but then `equals` compares the stored key — which is now `Spot(3,2)` — with `Spot(1,2)`, and says `false`. Again `null`.
- The entry has not gone anywhere: printing the map, which simply walks all buckets, shows `{Spot(3,2)=Siti's bag}`. The bag is in the rack, and no lookup can ever find it. We have created a memory leak and a lost bag with one assignment.
- The set suffers the same way: adding the "same" spot again succeeds, because the set looks in bucket 0, finds no equal element, and inserts a second one. The set now holds two equal elements, which a set is never supposed to do.

The fix is not a cleverer hash function; it is **immutability**. The record `Shelf` has `final` fields, so the key simply cannot change (the commented-out `shelf.row = 3` would not compile). To "move" a bag, we remove the entry under the old key and put it under a new key. **Listing 13.22** tells the same story in Python.

Listing 13.22 `mutable_key.py`: the same lost key in a `dict` and a `set`, the frozen-dataclass cure, and Python's refusal to hash a list.

```
"""mutable_key.py: what happens when a key changes while it is in a dict or set."""
from dataclasses import dataclass


class Spot:
    """A mutable point whose equality and hash come from its fields."""

    def __init__(self, row, col):
        self.row = row
        self.col = col

    def __eq__(self, other):
        return isinstance(other, Spot) and (self.row, self.col) == (other.row, other.col)

    def __hash__(self):
        return hash((self.row, self.col))

    def __repr__(self):
        return f"Spot({self.row},{self.col})"


@dataclass(frozen=True)
class Shelf:                                # frozen + eq  =>  a safe, hashable key
    row: int
    col: int


def main():
    spot = Spot(1, 2)
    rack = {spot: "Siti's bag"}
    used = {spot}
    before = hash(spot)
    spot.row = 3                            # mutate the key while it is in the dict
    print("hash changed:", before != hash(spot))
    print("rack.get(spot):          ", rack.get(spot))
    print("rack.get(Spot(1, 2)):    ", rack.get(Spot(1, 2)))
    print("spot in rack:            ", spot in rack)
    print("but the entry is still there:", rack)
    used.add(spot)
    print("used =", used, "size", len(used))
```

Listing 13.22 (Continued) *mutable_key.py: the same lost key in a dict and a set, the frozen-dataclass cure, and Python's refusal to hash a list.*

```
    shelf = Shelf(1, 2)
    shelves = {shelf: "Siti's bag"}
    moved = Shelf(3, 2)
    shelves[moved] = shelves.pop(shelf)
    print("shelves =", shelves)
    print("equal frozen objects, equal hashes:", hash(Shelf(3, 2)) == hash(moved))

    # Python refuses the classic mutable key outright.
    try:
        {[1, 2]: "bag"}
    except TypeError as e:
        print("list as key:", e)


if __name__ == "__main__":
    main()
```

```
hash changed: True
rack.get(spot):           None
rack.get(Spot(1, 2)):     None
spot in rack:             False
but the entry is still there: {Spot(3,2): "Siti's bag"}
used = {Spot(3,2), Spot(3,2)} size 2
shelves = {Shelf(row=3, col=2): "Siti's bag"}
equal frozen objects, equal hashes: True
list as key: unhashable type: 'list'
```

Python behaves identically, for identical reasons. It also takes a precaution that Java does not: the built-in mutable containers `list`, `dict` and `set` are deliberately **unhashable** — their `__hash__` is set to `None` — so the most common mutable keys are rejected immediately with `TypeError: unhashable type: 'list'`. A class that defines `__eq__` without `__hash__` becomes unhashable in the same way, as Chapter 7 showed. A frozen dataclass with `eq=True` (the default) gets a `__hash__` computed from its fields, which is exactly what a value object needs. Our `Spot` class, which defines both methods by hand on mutable fields, is precisely the dangerous combination.

> ✓ **KEY INSIGHT.** Use as keys (and as set elements) only objects whose equality **cannot change**: strings, numbers, dates, enums, records, frozen dataclasses, tuples of these. If an object must change, it is an entity; key it by something that does not change, such as an id.

13.10 Streams, Comprehensions, and itertools

So far, every time we wanted to compute something from a collection, we wrote a loop: create a result variable, walk the elements, test each one with `if`, update the result. That style says *how* to compute, step by step. Many computations over collections, however, are built from just three ideas:

- **filter** — keep only the elements that satisfy a condition;
- **map** — transform each element into something else;
- **reduce** (also called *fold*) — combine all the elements into one result.

"Revenue from loads of at least 4 kg at Rp7,000 per kilogram" is: filter the loads with $w \geq 4$, map each w to $7000\,w$, reduce with $+$. The reduce step starts from an initial value e (for a sum, 0) and combines it with each element in turn using an operation f:

$$\text{reduce}(f, e, [x_1, x_2, \ldots, x_n]) = f(\cdots f(f(e, x_1), x_2) \cdots, x_n) \tag{13.8}$$

For $f = +$ and $e = 0$, Equation (13.8) is simply $x_1 + x_2 + \cdots + x_n$. For $f = \max$ it is the largest element; for f = "join with a comma" it is a string. **Figure 13.5** shows the whole computation as a pipeline.

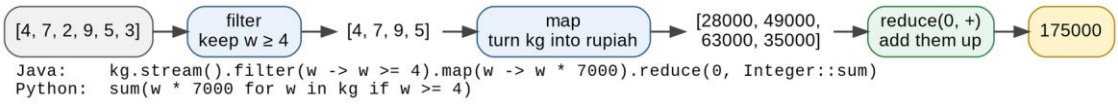


Figure 13.5 A filter–map–reduce pipeline. The source loads flow through a filter and a map into a reduction that folds them into one number; the same pipeline in one line of Java and one line of Python.

Java expresses such pipelines with **streams** (package `java.util.stream`, since Java 8). A *stream* is not a collection: it stores nothing. It is a sequence of elements flowing from a *source* (a collection, an array, a generator function) through any number of *intermediate operations* (`filter`, `map`, `sorted`, `distinct`, `limit`, …) to one *terminal operation* (`reduce`, `sum`, `count`, `collect`, `forEach`, `findFirst`, …) that produces the result [3]. Python expresses them with

comprehensions and **generator expressions**, the built-ins `map`, `filter`, `sum`, `min`, `max`, `any`, `all`, `functools.reduce`, and the module `itertools` [4]. **Listing 13.23** is a tour of the Java side.

Listing 13.23 `StreamsDemo.java`: a loop and a stream computing the same total, laziness made visible with `peek`, single use, collectors, and an infinite stream limited with `limit`.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class StreamsDemo {
    record Ticket(String customer, String service, int kg) { }

    public static void main(String[] args) {
        List<Integer> kg = List.of(4, 7, 2, 9, 5, 3);

        // The loop way: say HOW, step by step.
        int loopTotal = 0;
        for (int w : kg) {
            if (w >= 4) {
                loopTotal += w * 7000;
            }
        }
        System.out.println("loop:   " + loopTotal);

        // The stream way: say WHAT - keep, transform, combine.
        int streamTotal = kg.stream()
                .filter(w -> w >= 4)             // keep loads of at least 4 kg
                .map(w -> w * 7000)              // turn kilograms into rupiah
                .reduce(0, Integer::sum);        // fold them into one number
        System.out.println("stream: " + streamTotal);

        // Streams are lazy: nothing runs until a terminal operation asks.
        Stream<Integer> pipeline = kg.stream()
                .peek(w -> System.out.print(" saw " + w))
                .filter(w -> w > 6);
        System.out.println("pipeline built, nothing printed yet");
        System.out.println(" -> first heavy load: " + pipeline.findFirst().orElse(-1));

        // A stream can be used only once.
        Stream<Integer> once = kg.stream();
        once.count();
        try {
            once.count();
        } catch (IllegalStateException e) {
            System.out.println("second use: " + e.getMessage());
        }
```

Listing 13.23 (Continued) *StreamsDemo.java: a loop and a stream computing the same total, laziness made visible with peek, single use, collectors, and an infinite stream limited with limit.*

```
        // Collectors turn a stream back into a collection, a string or a map.
        List<Ticket> tickets = List.of(
                new Ticket("Siti", "WF", 4), new Ticket("Budi", "IR", 2),
                new Ticket("Siti", "IR", 3), new Ticket("Ayu", "WF", 6),
                new Ticket("Budi", "WF", 5));
        List<String> heavy = tickets.stream()
                .filter(t -> t.kg() >= 4)
                .map(Ticket::customer)
                .distinct()
                .sorted()
                .toList();                          // Java 16+; an unmodifiable list
        System.out.println("heavy customers: " + heavy);
        Map<String, Integer> kgPerService = tickets.stream()
                .collect(Collectors.groupingBy(Ticket::service, TreeMap::new,
                        Collectors.summingInt(Ticket::kg)));
        System.out.println("kg per service: " + kgPerService);
        System.out.println("names: " + tickets.stream().map(Ticket::customer)
                .collect(Collectors.joining(", ", "<", ">")));

        // Infinite streams are fine, as long as something limits them.
        List<Integer> ticketNumbers = Stream.iterate(101, n -> n + 1)
                .filter(n -> n % 10 != 4)           // skip numbers ending in 4
                .limit(5)
                .collect(Collectors.toCollection(ArrayList::new));
        System.out.println("ticket numbers: " + ticketNumbers);
        System.out.println("1 + 2 + ... + 100 = " + IntStream.rangeClosed(1, 100).sum());
    }
}
```

```
loop:   175000
stream: 175000
pipeline built, nothing printed yet
 saw 4 saw 7 -> first heavy load: 7
second use: stream has already been operated upon or closed
heavy customers: [Ayu, Budi, Siti]
kg per service: {IR=5, WF=15}
names: <Siti, Budi, Siti, Ayu, Budi>
ticket numbers: [101, 102, 103, 105, 106]
1 + 2 + ... + 100 = 5050
```

- **Loop versus stream.** Both compute Rp175,000 = 7000 × (4 + 7 + 9 + 5). The stream reads top to bottom as the sentence we started from: `filter(w -> w >= 4)`, `map(w -> w * 7000)`, `reduce(0, Integer::sum)`. The arguments are *lambda expressions* — small anonymous functions, written `parameter -> result` — and *method references* such as `Integer::sum`, which means "the method `sum` of class `Integer`", a function that adds two `int`s. Lambdas and streams arrived together in Java 8, and they were designed to be used together [18].
- **Laziness.** The second stream contains `peek`, which lets us watch elements pass. Building the pipeline prints nothing: intermediate operations only *describe* the work. When the terminal operation `findFirst()` runs, elements flow through one at a time — 4 is seen and rejected, 7 is seen and accepted — and the stream **stops**, because `findFirst` has its answer. The elements 2, 9, 5 and 3 are never examined. (The words "saw 4 saw 7" appear before "first heavy load" because Java evaluates the whole argument of `println`, including `findFirst()`, before printing it.)
- **Single use.** A stream, like an iterator, can be walked only once. The second `count()` throws `IllegalStateException: stream has already been operated upon or closed`. To run another query, ask the collection for a new stream.

- **Collectors.** `toList()` (Java 16+) collects into an unmodifiable list; `distinct()` and `sorted()` remove duplicates and order the result. `Collectors.groupingBy(Ticket::service, TreeMap::new, Collectors.summingInt(Ticket::kg))` builds a map from each service code to its total kilograms — the "group by" of a database, in one expression — and `Collectors.joining(", ", "<", ">")` reduces strings into one.
- **Infinite streams.** `Stream.iterate(101, n -> n + 1)` is an endless sequence 101, 102, 103, ...; `filter` drops the ticket numbers ending in 4, and `limit(5)` cuts the stream after five elements. Because streams are lazy, the infinite source is no problem: only six numbers (101 to 106) are ever generated. `IntStream.rangeClosed(1, 100).sum()` is the sum $1 + 2 + \cdots + 100 = 5050$, a stream of primitive `int`s that avoids boxing.

Listing 13.24 is the Python side of the same tour.

Listing 13.24 `streams_demo.py`: comprehensions and generator expressions, `map`, `filter` and `reduce`, laziness, single use, grouping, and infinite iterators from `itertools`.

```
"""streams_demo.py: comprehensions, generator expressions, reduce and itertools."""
from collections import namedtuple
from functools import reduce
from itertools import accumulate, count, groupby, islice
from operator import add, attrgetter

Ticket = namedtuple("Ticket", "customer service kg")


def main():
    kg = [4, 7, 2, 9, 5, 3]

    # The loop way: say HOW, step by step.
    loop_total = 0
    for w in kg:
        if w >= 4:
            loop_total += w * 7000
    print("loop:  ", loop_total)

    # The declarative way: say WHAT - keep, transform, combine.
    print("comp:  ", sum(w * 7000 for w in kg if w >= 4))
    print("reduce:", reduce(add, map(lambda w: w * 7000, filter(lambda w: w >= 4, kg)), 0))

    # Generator expressions are lazy; list comprehensions are not.
    def seen(w):
        print(f" saw {w}", end="")
        return w

    pipeline = (w for w in map(seen, kg) if w > 6)
    print("pipeline built, nothing printed yet")
    print(" -> first heavy load:", next(pipeline, -1))

    # A generator can be used only once.
    once = (w for w in kg)
    print("first use:", sum(1 for _ in once), " second use:", sum(1 for _ in once))

    tickets = [Ticket("Siti", "WF", 4), Ticket("Budi", "IR", 2), Ticket("Siti", "IR", 3),
               Ticket("Ayu", "WF", 6), Ticket("Budi", "WF", 5)]
    heavy = sorted({t.customer for t in tickets if t.kg >= 4})    # a set comprehension
    print("heavy customers:", heavy)
    by_service = sorted(tickets, key=attrgetter("service"))       # groupby needs sorted
input
    kg_per_service = {s: sum(t.kg for t in group)
                      for s, group in groupby(by_service, key=attrgetter("service"))}
    print("kg per service:", kg_per_service)
    print("names:", "<" + ", ".join(t.customer for t in tickets) + ">")
```

Listing 13.24 (Continued) *streams_demo.py: comprehensions and generator expressions, map, filter and reduce, laziness, single use, grouping, and infinite iterators from itertools.*

```
    # Infinite iterators are fine, as long as something limits them.
    ticket_numbers = list(islice((n for n in count(101) if n % 10 != 4), 5))
    print("ticket numbers:", ticket_numbers)
    print("1 + 2 + ... + 100 =", sum(range(1, 101)))
    print("running total of kg:", list(accumulate(kg)))


if __name__ == "__main__":
    main()
```

```
loop:   175000
comp:   175000
reduce: 175000
pipeline built, nothing printed yet
 saw 4 saw 7 -> first heavy load: 7
first use: 6  second use: 0
heavy customers: ['Ayu', 'Budi', 'Siti']
kg per service: {'IR': 5, 'WF': 15}
names: <Siti, Budi, Siti, Ayu, Budi>
ticket numbers: [101, 102, 103, 105, 106]
1 + 2 + ... + 100 = 5050
running total of kg: [4, 11, 13, 22, 27, 30]
```

- **Comprehensions.** `sum(w * 7000 for w in kg if w >= 4)` reads almost like the mathematical $\sum_{w \in kg,\, w \geq 4} 7000\ w$. The part in parentheses is a **generator expression**: the form `expression for variable in iterable if condition` is a map (the expression), a source (the `for`) and a filter (the `if`) in one. Put it in square brackets and it becomes a **list comprehension** that builds a list; in braces, a **set comprehension** (`{t.customer for t in tickets if t.kg >= 4}`) or, with `key: value`, a **dictionary comprehension**.
- **map, filter, reduce.** The line labeled `reduce:` writes the same pipeline with the built-in functions `map` and `filter` and with `functools.reduce(add, ..., 0)`, which is Equation (13.8) with f = `operator.add` and $e = 0$. It works, but most Python programmers prefer the comprehension, which needs no lambdas.
- **Laziness.** A generator expression is lazy, like a Java stream: building `pipeline` prints nothing, and `next(pipeline, -1)` pulls elements through `seen` only until the first one above 6. A list comprehension, by contrast, is **eager**: it runs completely and builds the whole list at once. Use brackets when you need the list; use parentheses when you only need to walk the results once, especially over large or infinite sources.
- **Single use.** A generator expression is an iterator, so it is used up after one walk: the second `sum` counts 0 elements. (Unlike Java's stream, it does not throw; it is simply empty — the trap of Section 13.4.3.)
- **Grouping.** `itertools.groupby` groups *consecutive* equal keys, so the tickets are first sorted by service; the dictionary comprehension then sums the kilograms in each group. (`collections.Counter` and `defaultdict` are other common tools for grouping.)
- **Infinite iterators.** `itertools.count(101)` yields 101, 102, ... forever; the generator expression filters it, and `itertools.islice(..., 5)` takes five elements — the Python spelling of `Stream.iterate(...).filter(...).limit(5)`. `itertools.accumulate(kg)` yields running totals, the intermediate values of Equation (13.8).

Table 13.4 is a phrase book between the two languages.

Table 13.4 Stream operations in Java and their usual Python equivalents.

Operation	Java stream	Python
Keep matching elements	`.filter(x -> p(x))`	`(x for x in xs if p(x))`, `filter(p, xs)`
Transform each element	`.map(x -> f(x))`	`(f(x) for x in xs)`, `map(f, xs)`
Flatten nested collections	`.flatMap(x -> x.items().stream())`	`(y for x in xs for y in x.items)`, `itertools.chain.from_iterable`
Fold into one value	`.reduce(e, f)`	`functools.reduce(f, xs, e)`
Sum, count, min, max	`.mapToInt(...).sum()`, `.count()`, `.min(c)`, `.max(c)`	`sum`, `len(list)`, `min(xs, key=...)`, `max(...)`
Any / all match	`.anyMatch(p)`, `.allMatch(p)`	`any(p(x) for x in xs)`, `all(...)`
First match	`.filter(p).findFirst()`	`next((x for x in xs if p(x)), default)`
Take the first *k*	`.limit(k)`	`itertools.islice(xs, k)`
Remove duplicates, sort	`.distinct()`, `.sorted(c)`	`set(...)` / `dict.fromkeys(...)`, `sorted(xs, key=...)`
Group	`Collectors.groupingBy(key, collector)`	`itertools.groupby` on sorted data, `Counter`, `defaultdict`
Collect into a list	`.toList()` (unmodifiable), `Collectors.toList()`	`[... for ...]`, `list(...)`
Infinite source	`Stream.iterate(seed, f)`, `Stream.generate(s)`	`itertools.count`, `itertools.cycle`, generator functions

A last word of advice. Pipelines are at their best when each step is simple and has **no side effects**: `filter` only tests, `map` only computes (our `peek` was for teaching only). When a loop is clearer — several branches, complicated early exits — write the loop. Clarity, not fashion, decides.

13.11 Profitina Laundry: Money, an Order History, and a Report Pipeline

Time to put the chapter together. The laundry needs three things.

1. **Money and prices that cannot be changed by accident.** An amount of money is a value object. A price rule — Wash & Fold at Rp7,000 per kilogram, minimum 3 kg — is a value object too. Once an order is recorded, its lines and total must be frozen.
2. **An order history** that can be walked newest-first for the counter screen, and filtered to one customer's orders for the question "what did Siti bring last week?" — both lazily, without copying the history.
3. **A report** built as a pipeline: revenue per service, the top three customers, kilograms washed per day, the average order, and the list of large orders.

The price rules are those of Chapter 7: Wash & Fold (`WF`) Rp7,000 per kilogram with a 3 kg minimum, Iron Only (`IR`) Rp5,000 with a 2 kg minimum, Wash, Dry & Iron (`WDI`) Rp10,000 with a 3 kg minimum, and members pay 10% less, the discount rounded down. The chargeable weight is the actual weight raised to the minimum and rounded up to the next half kilogram. To keep every calculation exact, this time we store weights as whole numbers of **tenths of a kilogram** (4.2 kg is 42) and money as whole rupiah in a `long` — never in a `double`, whose binary fractions cannot

represent most decimal amounts exactly. With w and $w_{\min}$ in tenths, the chargeable weight in tenths is

$$\hat{w} = 5\left\lceil \frac{\max(w, w_{\min})}{5} \right\rceil \tag{13.9}$$

and the price for a rate r per kilogram is $r\hat{w}/10$. For 4.2 kg of Wash & Fold, $\max(42,30) = 42$, $\lceil 42/5 \rceil = 9$, so $\hat{w} = 45$ tenths, and the price is $7000 \cdot 45/10 = 31{,}500$ rupiah. In integer arithmetic, $\lceil a/5 \rceil$ is computed as `(a + 4) / 5`. Equation (13.9) is the same rule as Chapter 7's, in whole numbers.

Figure 13.6 is the design. Four records are value objects: `Money`, `Price`, `OrderLine` and `Order`. `OrderHistory` is an immutable, append-only collection that implements `Iterable<Order>`, and its iterators are inner classes.

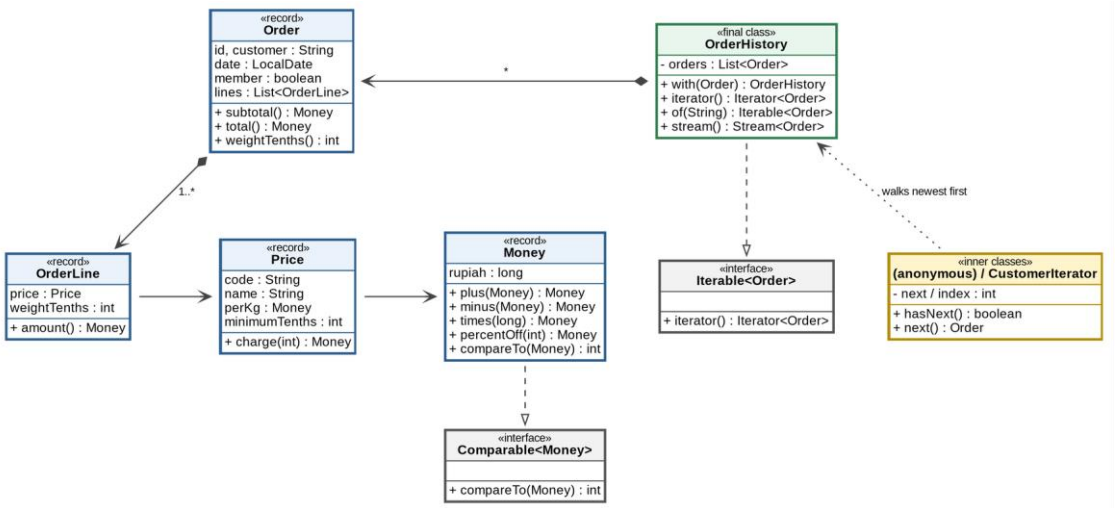


Figure 13.6 UML class diagram of the laundry ledger. `Money`, `Price`, `OrderLine` and `Order` are records (immutable values); `OrderHistory` holds an unmodifiable list of orders, implements `Iterable<Order>`, and hands out two kinds of iterator.

Listing 13.25 is the complete program.

Listing 13.25 `LaundryLedger.java`: Profitina Laundry with an immutable `Money` record, price rules, frozen orders, an append-only `OrderHistory` with two custom iterators, and a report built from streams.

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.NoSuchElementException;
import java.util.TreeMap;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class LaundryLedger {
    static final Price WF = new Price("WF", "Wash & Fold", Money.of(7_000), 30);
    static final Price IR = new Price("IR", "Iron Only", Money.of(5_000), 20);
    static final Price WDI = new Price("WDI", "Wash, Dry & Iron", Money.of(10_000), 30);

    static OrderHistory septemberHistory() {
        LocalDate d21 = LocalDate.of(2026, 9, 21);
        LocalDate d22 = d21.plusDays(1);
        LocalDate d23 = d21.plusDays(2);
        return OrderHistory.EMPTY
            .with(new Order("ORD-1301", "Siti Aminah", d21, true,
                    List.of(new OrderLine(WF, 42), new OrderLine(IR, 13))))
            .with(new Order("ORD-1302", "Budi Santoso", d21, false,
                    List.of(new OrderLine(WDI, 50))))
            .with(new Order("ORD-1303", "Ayu Lestari", d21, false,
                    List.of(new OrderLine(WF, 20))))
            .with(new Order("ORD-1304", "Siti Aminah", d22, true,
                    List.of(new OrderLine(WDI, 34))))
            .with(new Order("ORD-1305", "Dewi Anggraini", d22, false,
                    List.of(new OrderLine(WF, 79), new OrderLine(IR, 25))))
            .with(new Order("ORD-1306", "Budi Santoso", d22, false,
                    List.of(new OrderLine(IR, 30))))
            .with(new Order("ORD-1307", "Ayu Lestari", d23, false,
                    List.of(new OrderLine(WF, 51), new OrderLine(WDI, 22))))
            .with(new Order("ORD-1308", "Siti Aminah", d23, true,
                    List.of(new OrderLine(WF, 60))));
    }

    public static void main(String[] args) {
        OrderHistory history = septemberHistory();
```

Listing 13.25 (Continued) *LaundryLedger.java: Profitina Laundry with an immutable Money record, price rules, frozen orders, an append-only OrderHistory with two custom iterators, and a report built from streams.*

```
        System.out.println("== Order history, newest first ==");
        for (Order o : history) {                  // uses OrderHistory.iterator()
            System.out.println(o.summary());
        }

        System.out.println("== Siti Aminah's orders ==");
        for (Order o : history.of("Siti Aminah")) {
            System.out.println(o.summary());
        }

        System.out.println("== Revenue per service ==");
        Map<String, Money> perService = history.stream()
                .flatMap(o -> o.lines().stream())
                .collect(Collectors.groupingBy(l -> l.price().code(), TreeMap::new,
                        Collectors.reducing(Money.ZERO, OrderLine::amount, Money::plus)));
        perService.forEach((code, money) ->
                System.out.printf(Locale.US, "%-4s %12s%n", code, money));

        System.out.println("== Top three customers ==");
        Map<String, Money> perCustomer = history.stream()
                .collect(Collectors.groupingBy(Order::customer,
                        Collectors.reducing(Money.ZERO, Order::total, Money::plus)));
        perCustomer.entrySet().stream()
                .sorted(Map.Entry.<String, Money>comparingByValue().reversed()
                        .thenComparing(Map.Entry.comparingByKey()))
                .limit(3)
                .forEach(e -> System.out.printf(Locale.US, "%-15s %10s%n",
                        e.getKey(), e.getValue()));

        System.out.println("== Kilograms per day ==");
        Map<LocalDate, Integer> tenthsPerDay = history.stream()
                .collect(Collectors.groupingBy(Order::date, TreeMap::new,
                        Collectors.summingInt(Order::weightTenths)));
        tenthsPerDay.forEach((day, tenths) ->
                System.out.printf(Locale.US, "%s %5.1f kg%n", day, tenths / 10.0));

        Money revenue = history.stream().map(Order::total).reduce(Money.ZERO, Money::plus);
        long orders = history.stream().count();
        System.out.println("Revenue " + revenue + " from " + orders + " orders, average "
                + Money.of(revenue.rupiah() / orders));
        System.out.println("Orders of Rp50,000 or more: " + history.stream()
                .filter(o -> o.total().compareTo(Money.of(50_000)) >= 0)
                .map(Order::id)
                .toList());
```

Listing 13.25 (Continued) *LaundryLedger.java: Profitina Laundry with an immutable Money record, price rules, frozen orders, an append-only OrderHistory with two custom iterators, and a report built from streams.*

```
        System.out.println("== Immutability at work ==");
        OrderHistory longer = history.with(new Order("ORD-1309", "Budi Santoso",
                LocalDate.of(2026, 9, 23), false, List.of(new OrderLine(IR, 21))));
        System.out.println("old history: " + history.size() + " orders, new history: "
                + longer.size() + " orders");
        Order first = history.iterator().next();
        try {
            first.lines().add(new OrderLine(WF, 10));
        } catch (UnsupportedOperationException e) {
            System.out.println("adding a line to a stored order: "
                    + e.getClass().getSimpleName());
        }
        try {
            Money.of(5_000).minus(Money.of(7_000));
        } catch (IllegalArgumentException e) {
            System.out.println("Rp5,000 - Rp7,000: " + e.getMessage());
        }
    }
}

// An amount of rupiah. Immutable: every operation returns a new Money.
record Money(long rupiah) implements Comparable<Money> {
    static final Money ZERO = new Money(0);

    Money {
        if (rupiah < 0) {
            throw new IllegalArgumentException("money cannot be negative");
        }
    }

    static Money of(long rupiah) {
        return new Money(rupiah);
    }

    Money plus(Money other) {
        return new Money(Math.addExact(rupiah, other.rupiah));
    }

    Money minus(Money other) {
        return new Money(rupiah - other.rupiah);   // the constructor rejects < 0
    }

    Money times(long factor) {
        return new Money(Math.multiplyExact(rupiah, factor));
    }
```

Listing 13.25 (Continued) *LaundryLedger.java: Profitina Laundry with an immutable Money record, price rules, frozen orders, an append-only OrderHistory with two custom iterators, and a report built from streams.*

```
    Money percentOff(int percent) {
        return new Money(rupiah - rupiah * percent / 100);  // discount rounded down
    }

    @Override
    public int compareTo(Money other) {
        return Long.compare(rupiah, other.rupiah);
    }

    @Override
    public String toString() {
        return String.format(Locale.US, "Rp%,d", rupiah);
    }
}

// A price rule: rate per kilogram and a minimum, weights in tenths of a kilogram.
record Price(String code, String name, Money perKg, int minimumTenths) {
    Money charge(int weightTenths) {
        int w = Math.max(weightTenths, minimumTenths);
        int chargeable = (w + 4) / 5 * 5;          // round up to the next 0.5 kg
        return Money.of(perKg.rupiah() * chargeable / 10);
    }
}

record OrderLine(Price price, int weightTenths) {
    OrderLine {
        if (weightTenths <= 0) {
            throw new IllegalArgumentException("weight must be positive");
        }
    }

    Money amount() {
        return price.charge(weightTenths);
    }
}

record Order(String id, String customer, LocalDate date, boolean member,
             List<OrderLine> lines) {
    Order {
        lines = List.copyOf(lines);                // defensive, unmodifiable copy
    }

    Money subtotal() {
        return lines.stream().map(OrderLine::amount).reduce(Money.ZERO, Money::plus);
    }
```

Listing 13.25 (Continued) *LaundryLedger.java: Profitina Laundry with an immutable Money record, price rules, frozen orders, an append-only OrderHistory with two custom iterators, and a report built from streams.*

```
    Money total() {
        return member ? subtotal().percentOff(10) : subtotal();
    }

    int weightTenths() {
        return lines.stream().mapToInt(OrderLine::weightTenths).sum();
    }

    String summary() {
        return String.format(Locale.US, "%s %s %-15s %10s", id, date, customer, total());
    }
}

// An append-only history. with() returns a NEW history; this one never changes.
final class OrderHistory implements Iterable<Order> {
    static final OrderHistory EMPTY = new OrderHistory(List.of());
    private final List<Order> orders;

    private OrderHistory(List<Order> orders) {
        this.orders = List.copyOf(orders);
    }

    OrderHistory with(Order order) {
        List<Order> longer = new ArrayList<>(orders);
        longer.add(order);
        return new OrderHistory(longer);
    }

    int size() {
        return orders.size();
    }

    Stream<Order> stream() {
        return orders.stream();
    }

    // Newest first: the iterator walks the list from its last index down to 0.
    @Override
    public Iterator<Order> iterator() {
        return new Iterator<>() {
            private int next = orders.size() - 1;
```

Listing 13.25 (Continued) *LaundryLedger.java: Profitina Laundry with an immutable Money record, price rules, frozen orders, an append-only OrderHistory with two custom iterators, and a report built from streams.*

```
            @Override
            public boolean hasNext() {
                return next >= 0;
            }

            @Override
            public Order next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                return orders.get(next--);
            }
        };
    }

    // Only one customer's orders, oldest first, found lazily one at a time.
    Iterable<Order> of(String customer) {
        return () -> new CustomerIterator(customer);
    }

    private final class CustomerIterator implements Iterator<Order> {
        private final String customer;
        private int index = 0;

        CustomerIterator(String customer) {
            this.customer = customer;
            skipOthers();
        }

        private void skipOthers() {
            while (index < orders.size() && !orders.get(index).customer().equals(customer))
{
                index++;
            }
        }

        @Override
        public boolean hasNext() {
            return index < orders.size();
        }
```

Listing 13.25 (Continued) *LaundryLedger.java: Profitina Laundry with an immutable Money record, price rules, frozen orders, an append-only OrderHistory with two custom iterators, and a report built from streams.*

```
        @Override
        public Order next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            Order found = orders.get(index++);
            skipOthers();                          // look ahead to the next match
            return found;
        }
    }
}
```

```
== Order history, newest first ==
ORD-1308 2026-09-23 Siti Aminah       Rp37,800
ORD-1307 2026-09-23 Ayu Lestari       Rp68,500
ORD-1306 2026-09-22 Budi Santoso      Rp15,000
ORD-1305 2026-09-22 Dewi Anggraini    Rp68,500
ORD-1304 2026-09-22 Siti Aminah       Rp31,500
ORD-1303 2026-09-21 Ayu Lestari       Rp21,000
ORD-1302 2026-09-21 Budi Santoso      Rp50,000
ORD-1301 2026-09-21 Siti Aminah       Rp37,350
== Siti Aminah's orders ==
ORD-1301 2026-09-21 Siti Aminah       Rp37,350
ORD-1304 2026-09-22 Siti Aminah       Rp31,500
ORD-1308 2026-09-23 Siti Aminah       Rp37,800
== Revenue per service ==
IR       Rp37,500
WDI     Rp115,000
WF      Rp189,000
== Top three customers ==
Siti Aminah      Rp106,650
Ayu Lestari       Rp89,500
Dewi Anggraini    Rp68,500
== Kilograms per day ==
2026-09-21  12.5 kg
2026-09-22  16.8 kg
2026-09-23  13.3 kg
Revenue Rp329,650 from 8 orders, average Rp41,206
Orders of Rp50,000 or more: [ORD-1302, ORD-1305, ORD-1307]
== Immutability at work ==
old history: 8 orders, new history: 9 orders
adding a line to a stored order: UnsupportedOperationException
Rp5,000 - Rp7,000: money cannot be negative
```

Let us read it from the smallest part upwards.

`Money` is a record holding a `long` number of rupiah. Its compact constructor enforces the invariant "money is never negative", so no `Money` object can ever hold a negative amount — not at construction, not after any calculation, because every calculation builds a new `Money` through the same constructor. That is why `minus` needs no check of its own: `Money.of(5_000).minus(Money.of(7_000))` tries to build −2000 and the constructor refuses, as the last line of output shows. `plus` and `times` use `Math.addExact` and `Math.multiplyExact`, which throw on overflow instead of wrapping around as in Equation (13.5). `percentOff(10)` subtracts `rupiah * 10 / 100`, the discount rounded down, the rule of Chapter 7. `Money` implements `Comparable<Money>`, because amounts have one obvious natural order, and `toString` prints `Rp37,350`. Martin Fowler describes this *Money* pattern as one of the most useful value objects a business program can have [19].

`Price` is a record of a code, a display name, a rate as `Money`, and a minimum in tenths. Its `charge` method is Equation (13.9) in code. `OrderLine` pairs a `Price` with a weight and rejects non-positive weights. `Order` holds an id, a customer name, a `LocalDate` (itself an immutable value), a membership flag and a list of lines; its compact constructor replaces the list with `List.copyOf(lines)`, so an order, once built, can never gain or lose a line — the output confirms that `first.lines().add(...)` throws. `subtotal()` is a small stream: map each line to its amount, reduce with `Money::plus` starting from `Money.ZERO`.

`OrderHistory` is the heart of the chapter.

- It is **immutable and append-only**. Its constructor is private and copies the list with `List.copyOf`; `with(order)` does not change the history but returns a *new* history one order longer. The "Immutability at work" block shows that adding ORD-1309 to `history` left `history`

with 8 orders while `longer` has 9. This is the same `withX` pattern as `Point`, applied to a collection. Because each history is immutable, the report code, the counter screen and a background backup could all hold the same `OrderHistory` at once without any risk. (Copying the list on every `with` costs time proportional to its length; *persistent* data structures avoid that cost by sharing structure.)

- It is **iterable newest-first**. `iterator()` returns an *anonymous inner class* — `new Iterator<>() { ... }` creates an object of a class that has no name — whose cursor `next` starts at the last index and moves down. Chapter 10 used anonymous classes for listeners; here one serves as an iterator. Because `OrderHistory` is `Iterable<Order>`, the counter screen is one for-each loop: `for (Order o : history)`.
- It offers **a filtered view**: `of(customer)` returns an `Iterable<Order>` written as a lambda, `() -> new CustomerIterator(customer)`. `Iterable` has a single abstract method, so a lambda can implement it; each for-each loop over the result calls the lambda and gets a fresh `CustomerIterator`. That iterator uses **look-ahead**: its constructor and its `next()` both call `skipOthers()`, which moves `index` forward past orders of other customers, so that `hasNext()` can answer with one comparison. Nothing is copied: Siti's three orders are found one at a time, directly in the history's list.
- It offers **`stream()`**, so that the report can use the whole stream vocabulary.

The **report** in `main` is a sequence of pipelines, each answering one question:

- **Revenue per service** uses `flatMap(o -> o.lines().stream())` to turn orders into all their lines, then groups by price code into a `TreeMap` (sorted keys), reducing each group with `Money::plus`. The per-service figures, Rp37,500 + Rp115,000 + Rp189,000 = Rp341,500, are line amounts *before* member discounts.
- **Top three customers** groups orders by customer and sums their *totals* (after discounts), then sorts the map's entries by value, descending, with ties broken by name, and keeps three with `limit(3)`. Siti comes first with Rp106,650.
- **Kilograms per day** groups by date and sums the weights with `summingInt`.
- **Revenue** reduces all totals: Rp329,650, which is Rp341,500 minus Siti's three member discounts of Rp4,150, Rp3,500 and Rp4,200. The average order is Rp329,650 / 8 = Rp41,206 (rounded down).
- **Large orders** filters by `compareTo(Money.of(50_000)) >= 0` and maps to ids. We compare `Money` objects with `compareTo`, not their fields: the value object decides its own order.

Listing 13.26 is the same system in Python.

Listing 13.26 `laundry_ledger.py`: the same ledger with frozen dataclasses, a generator-based newest-first iterator, a lazy per-customer generator, and a report built from comprehensions and `itertools.groupby`.

```
"""laundry_ledger.py: Profitina Laundry with immutable values, a history iterator
and a report pipeline."""
from dataclasses import dataclass
from datetime import date, timedelta
from itertools import groupby
from operator import attrgetter


@dataclass(frozen=True, order=True)
class Money:
    """An amount of rupiah. Immutable: every operation returns a new Money."""
    rupiah: int

    def __post_init__(self):
        if self.rupiah < 0:
            raise ValueError("money cannot be negative")

    def __add__(self, other):
        return Money(self.rupiah + other.rupiah)

    def __sub__(self, other):
        return Money(self.rupiah - other.rupiah)   # __post_init__ rejects < 0

    def __mul__(self, factor):
        return Money(self.rupiah * factor)

    def percent_off(self, percent):
        return Money(self.rupiah - self.rupiah * percent // 100)  # discount rounded down

    def __str__(self):
        return f"Rp{self.rupiah:,}"


ZERO = Money(0)


@dataclass(frozen=True)
class Price:
    """A price rule: rate per kilogram and a minimum, weights in tenths of a kilogram."""
    code: str
    name: str
    per_kg: Money
    minimum_tenths: int
```

Listing 13.26 (Continued) *laundry_ledger.py: the same ledger with frozen dataclasses, a generator-based newest-first iterator, a lazy per-customer generator, and a report built from comprehensions and itertools.groupby.*

```
    def charge(self, weight_tenths):
        w = max(weight_tenths, self.minimum_tenths)
        chargeable = (w + 4) // 5 * 5            # round up to the next 0.5 kg
        return Money(self.per_kg.rupiah * chargeable // 10)


@dataclass(frozen=True)
class OrderLine:
    price: Price
    weight_tenths: int

    def __post_init__(self):
        if self.weight_tenths <= 0:
            raise ValueError("weight must be positive")

    def amount(self):
        return self.price.charge(self.weight_tenths)


@dataclass(frozen=True)
class Order:
    id: str
    customer: str
    date: date
    member: bool
    lines: tuple

    def __post_init__(self):
        object.__setattr__(self, "lines", tuple(self.lines))   # defensive copy

    def subtotal(self):
        return sum((line.amount() for line in self.lines), ZERO)

    def total(self):
        return self.subtotal().percent_off(10) if self.member else self.subtotal()

    def weight_tenths(self):
        return sum(line.weight_tenths for line in self.lines)

    def summary(self):
        return f"{self.id} {self.date} {self.customer:<15} {str(self.total()):>10}"


class OrderHistory:
    """An append-only history. with_order() returns a NEW history."""
```

Listing 13.26 (Continued) *laundry_ledger.py: the same ledger with frozen dataclasses, a generator-based newest-first iterator, a lazy per-customer generator, and a report built from comprehensions and itertools.groupby.*

```
    def __init__(self, orders=()):
        self._orders = tuple(orders)

    def with_order(self, order):
        return OrderHistory(self._orders + (order,))

    def __len__(self):
        return len(self._orders)

    def __iter__(self):
        """Newest first: a generator that walks the tuple backwards."""
        for i in range(len(self._orders) - 1, -1, -1):
            yield self._orders[i]

    def of(self, customer):
        """Only one customer's orders, oldest first, found lazily one at a time."""
        return (o for o in self._orders if o.customer == customer)

    def oldest_first(self):
        return iter(self._orders)


WF = Price("WF", "Wash & Fold", Money(7_000), 30)
IR = Price("IR", "Iron Only", Money(5_000), 20)
WDI = Price("WDI", "Wash, Dry & Iron", Money(10_000), 30)


def september_history():
    d21 = date(2026, 9, 21)
    d22, d23 = d21 + timedelta(days=1), d21 + timedelta(days=2)
    history = OrderHistory()
    for order in [
        Order("ORD-1301", "Siti Aminah", d21, True, [OrderLine(WF, 42), OrderLine(IR,
13)]),
        Order("ORD-1302", "Budi Santoso", d21, False, [OrderLine(WDI, 50)]),
        Order("ORD-1303", "Ayu Lestari", d21, False, [OrderLine(WF, 20)]),
        Order("ORD-1304", "Siti Aminah", d22, True, [OrderLine(WDI, 34)]),
        Order("ORD-1305", "Dewi Anggraini", d22, False, [OrderLine(WF, 79), OrderLine(IR,
25)]),
        Order("ORD-1306", "Budi Santoso", d22, False, [OrderLine(IR, 30)]),
        Order("ORD-1307", "Ayu Lestari", d23, False, [OrderLine(WF, 51), OrderLine(WDI,
22)]),
        Order("ORD-1308", "Siti Aminah", d23, True, [OrderLine(WF, 60)]),
    ]:
        history = history.with_order(order)
    return history
```

Listing 13.26 (Continued) *laundry_ledger.py: the same ledger with frozen dataclasses, a generator-based newest-first iterator, a lazy per-customer generator, and a report built from comprehensions and itertools.groupby.*

```
def main():
    history = september_history()

    print("== Order history, newest first ==")
    for o in history:
        print(o.summary())

    print("== Siti Aminah's orders ==")
    for o in history.of("Siti Aminah"):
        print(o.summary())

    print("== Revenue per service ==")
    lines = [line for o in history.oldest_first() for line in o.lines]
    codes = sorted({line.price.code for line in lines})
    per_service = {code: sum((ln.amount() for ln in lines if ln.price.code == code), ZERO)
                   for code in codes}
    for code in codes:
        print(f"{code:<4} {str(per_service[code]):>12}")

    print("== Top three customers ==")
    by_customer = sorted(history.oldest_first(), key=attrgetter("customer"))
    per_customer = {name: sum((o.total() for o in group), ZERO)
                    for name, group in groupby(by_customer, key=attrgetter("customer"))}
    ranking = sorted(per_customer.items(), key=lambda kv: (-kv[1].rupiah, kv[0]))
    for name, money in ranking[:3]:
        print(f"{name:<15} {str(money):>10}")

    print("== Kilograms per day ==")
    for day, group in groupby(history.oldest_first(), key=attrgetter("date")):
        tenths = sum(o.weight_tenths() for o in group)
        print(f"{day} {tenths / 10:5.1f} kg")

    revenue = sum((o.total() for o in history), ZERO)
    count = len(history)
    print(f"Revenue {revenue} from {count} orders, average {Money(revenue.rupiah //
count)}")
    print("Orders of Rp50,000 or more:",
          [o.id for o in history.oldest_first() if o.total() >= Money(50_000)])
```

Listing 13.26 (Continued) *laundry_ledger.py: the same ledger with frozen dataclasses, a generator-based newest-first iterator, a lazy per-customer generator, and a report built from comprehensions and itertools.groupby.*

```
    print("== Immutability at work ==")
    longer = history.with_order(Order("ORD-1309", "Budi Santoso", date(2026, 9, 23), False,
                                      [OrderLine(IR, 21)]))
    print(f"old history: {len(history)} orders, new history: {len(longer)} orders")
    first = next(iter(history))
    try:
        first.lines.append(OrderLine(WF, 10))
    except AttributeError as e:
        print("adding a line to a stored order:", type(e).__name__)
    try:
        Money(5_000) - Money(7_000)
    except ValueError as e:
        print("Rp5,000 - Rp7,000:", e)


if __name__ == "__main__":
    main()
```

```
== Order history, newest first ==
ORD-1308 2026-09-23 Siti Aminah       Rp37,800
ORD-1307 2026-09-23 Ayu Lestari       Rp68,500
ORD-1306 2026-09-22 Budi Santoso      Rp15,000
ORD-1305 2026-09-22 Dewi Anggraini    Rp68,500
ORD-1304 2026-09-22 Siti Aminah       Rp31,500
ORD-1303 2026-09-21 Ayu Lestari       Rp21,000
ORD-1302 2026-09-21 Budi Santoso      Rp50,000
ORD-1301 2026-09-21 Siti Aminah       Rp37,350
== Siti Aminah's orders ==
ORD-1301 2026-09-21 Siti Aminah       Rp37,350
ORD-1304 2026-09-22 Siti Aminah       Rp31,500
ORD-1308 2026-09-23 Siti Aminah       Rp37,800
== Revenue per service ==
IR       Rp37,500
WDI     Rp115,000
WF      Rp189,000
== Top three customers ==
Siti Aminah      Rp106,650
Ayu Lestari       Rp89,500
Dewi Anggraini    Rp68,500
== Kilograms per day ==
2026-09-21  12.5 kg
2026-09-22  16.8 kg
2026-09-23  13.3 kg
Revenue Rp329,650 from 8 orders, average Rp41,206
Orders of Rp50,000 or more: ['ORD-1302', 'ORD-1305', 'ORD-1307']
== Immutability at work ==
old history: 8 orders, new history: 9 orders
adding a line to a stored order: AttributeError
Rp5,000 - Rp7,000: money cannot be negative
```

The output matches Java's apart from the printing of a list of strings and the exception's name. The differences in the code are instructive.

- `Money` is `@dataclass(frozen=True, order=True)`: frozen for immutability, `order=True` for the natural order (it compares the one field, `rupiah`). Instead of methods named `plus` and `minus` it defines the operators `__add__`, `__sub__` and `__mul__`, so amounts are written `a + b`. And because `Money` supports `+`, the built-in `sum` works on it when given a starting value: `sum((line.amount() for line in self.lines), ZERO)` is Equation (13.8) with e = `ZERO`.
- `Order.__post_init__` turns the lines into a tuple, the defensive copy of Section 13.8.3.
- `OrderHistory` stores a tuple, and `with_order` returns a new history built from `self._orders + (order,)`. Its `__iter__` is a **generator** that walks the indexes backwards — the whole Java anonymous iterator class in two lines. `of(customer)` returns a **generator expression**, which is lazy exactly like Java's `CustomerIterator`. `oldest_first()` returns a plain iterator over the tuple for the report.
- The report uses a set comprehension to find the service codes, dictionary comprehensions to total them, `sorted` with a tuple key `(-money.rupiah, name)` for "largest first, then by name", and `groupby` for per-customer and per-day totals. The orders are already in date order, so grouping by date needs no sort; grouping by customer does.

13.12 Profitina in Practice

The laundry's ledger is small, but its three ideas carry straight over to Profitina, the AI visibility (GEO) platform whose scanner measures how ready a website is to be cited by AI answer engines.

Scan results as values. When Profitina scans a site, each signal — content extractability, structured data, `robots.txt` access for answer engines and so on — produces a measurement and

a score with an explanation. Once a scan is finished, its result is a *fact about the past*: the site looked like this on that day. Such results are naturally immutable value objects, like `Order` in **Listing 13.25**. That is what makes the **historical analytics dashboard** trustworthy: the weekly score trend is computed from results that no later code can edit, the same week always shows the same number, and the CSV export writes exactly what the chart shows.

Walking large histories lazily. The trend for a brand is a walk over a growing history of scans, newest first, filtered to one brand — the `OrderHistory` iterators at a larger scale. A lazy walk shows the latest weeks at once without loading years of results.

Reports as pipelines. The dashboard's figures — average score per week, the signals that improved most — are filter–map–reduce pipelines over immutable scan results, written with comprehensions in Profitina's Python code base, like the report in **Listing 13.26**.

Immutability and personal data. Profitina complies with Indonesia's personal data protection law (UU PDP No. 27/2022): consent is explicit, and users can export or purge their data. Code that handles personal data benefits from the defensive copies of Section 13.8: an export receives an immutable snapshot, so what the user downloads is exactly what was stored at that moment. As always, the dashboard's numbers are advice: the platform explains what it measured; the user decides what to change.

13.13 Common Mistakes, and How to Avoid Them

1. Calling `next()` twice in one loop body. `while (it.hasNext()) { if (it.next().isBig()) list.add(it.next()); }` tests one element and adds the next one — and may throw `NoSuchElementException` at the end. *Avoid it:* call `next()` exactly once per iteration and store the result in a variable.

2. Changing a collection's size inside a for-each loop. In Java this usually throws `ConcurrentModificationException` and sometimes silently skips an element (block 4 of **Listing 13.7**); in Python it silently skips elements in a list and raises `RuntimeError` for a dict or set. *Avoid it:* use `Iterator.remove()`, `removeIf`, a comprehension, or a loop over a copy (**Table 13.2**).

3. Walking a Python iterator twice. A generator or the result of `map`, `filter`, `zip` or `iter(...)` is empty the second time, without any error. *Avoid it:* keep an iterable (a list, a tuple, an object with `__iter__`), or call `list(...)` once and reuse the list.

4. Writing `compareTo` or `compare` that returns only −1, 0, 1 — or assuming it does. Comparison methods may return any `int`; only the sign matters. Code that `switch`es on −1 and 1 breaks the contract of Equation (13.3), and the "sorted" result depends on the input order (**Listing 13.11**). *Avoid it:* pass the result of `compareTo` on unchanged, or build comparators with `Comparator.comparing(...).thenComparing(...)` (Python: tuple keys).

5. Comparing by subtraction. `return a - b;` can overflow (Equation (13.5)). *Avoid it:* `Integer.compare(a, b)`, `Long.compare`, or `Comparator.comparingInt`.

6. Thinking a shallow copy is independent. `new ArrayList<>(people)` and `list(people)` copy the list, not the persons in it. *Avoid it:* use a deep copy (copy constructors in Java, `copy.deepcopy` in Python) when elements are mutable — or, better, make the elements immutable, and then the shallow copy is enough.

7. Believing `final` makes an object immutable. A `final List` can still be changed; only the reference is fixed. *Avoid it:* use `List.copyOf` or `List.of` for the contents, and remember that even an unmodifiable list is only as immutable as its elements.

8. Confusing an unmodifiable view with an immutable copy. `Collections.unmodifiableList(x)` and `MappingProxyType(d)` show later changes to `x` or `d`. *Avoid it:* hand out a view when others should see live data; store a copy (`List.copyOf`, `tuple(...)`) when *you* need data that never changes.

9. Storing or returning a mutable field without a defensive copy. The `LeakyCustomer` of **Listing 13.19** was changed from outside twice. *Avoid it:* copy mutable arguments in the constructor (a record's compact constructor, a dataclass's `__post_init__`) and never return the internal object.

10. Mutating an object while it is a key in a map or an element of a set. The entry becomes unreachable (**Figure 13.4**). *Avoid it:* use immutable keys — strings, records, frozen dataclasses, tuples — and key entities by an unchanging id.

11. Putting side effects inside a stream or comprehension. `list.forEach(x -> other.add(...))` or a comprehension that appends to an outer list is a loop in disguise, with all a loop's risks and none of its clarity. *Avoid it:* let `map` and `filter` only compute, and let a collector or the comprehension itself build the result.

12. Reusing a stream. A Java stream throws `IllegalStateException` on its second terminal operation. *Avoid it:* call `collection.stream()` again for each query, or collect the stream into a list first.

13.14 Chapter Summary

- An **iterator** walks a collection one element at a time without revealing how it is stored. Java: `Iterator` with `hasNext()` and `next()` (Equation (13.1)); Python: `iter()` and `next()`, ending with `StopIteration` (**Table 13.1**). A plain iterator moves only forward and only reads; `ListIterator` can also move backwards, `set`, `add` and `remove`.
- An **iterable** produces iterators: Java's `Iterable` (which makes the for-each loop work), Python's `__iter__`. The for-each loop is a translated iterator loop. An iterable can be walked many times; an iterator only once.
- To write an iterable collection, keep the **collection** (what defines the data) apart from the **iterator** (the position of one walk); in Java, use a private inner class. In Python, a **generator** — a function with `yield` — writes the iterator for you, and runs lazily.
- Changing a collection's size while walking it is an error. Java's iterators detect it with `modCount` (Equation (13.2)) and throw `ConcurrentModificationException`, on a best-effort basis; Python lists skip elements silently, and dicts and sets raise `RuntimeError`. Use `Iterator.remove`, `removeIf`, comprehensions, or a copy (**Table 13.2**).
- `Comparable` gives a class its natural order; a `Comparator` is an order supplied from outside. A comparison must be antisymmetric and transitive (Equation (13.3), Equation (13.4)); only its sign matters; never subtract (Equation (13.5)). Python sorts with **key functions**, tuple keys for several criteria, and `cmp_to_key` for comparators; its sort, like Java's, is stable.
- **Aliasing** shares everything, a **shallow copy** shares the elements, a **deep copy** shares nothing (**Figure 13.3**). With immutable elements, a shallow copy suffices.

- An **immutable** object cannot change after construction. It is a safe key, a safe argument, safe to share between threads, and keeps its invariants for free. Rules: no mutators, a `final` class, `private final` fields, defensive copies of mutable parts, full construction. `final` fixes a reference, not an object; unmodifiable **views** show later changes, unmodifiable **copies** do not (**Table 13.3**). **Records** and **frozen dataclasses** make immutable value classes short.
- A **value object** is defined by its fields; equal values must have equal hashes (Equation (13.6)), because a hash table searches only the bucket of Equation (13.7). A key that changes while stored is lost (**Figure 13.4**); make keys immutable.
- **Streams** (Java) and **comprehensions, generator expressions and `itertools`** (Python) express computations as filter–map–reduce pipelines (Equation (13.8), **Figure 13.5**, **Table 13.4**). They are lazy, single-use, and best without side effects.
- In the laundry, `Money`, `Price`, `OrderLine` and `Order` are immutable values, `OrderHistory` is an immutable iterable with newest-first and per-customer iterators, and the report is a set of pipelines (**Figure 13.6**).

13.15 Exercises

Exercise 13.1 (tracing an iterator). For the list `[10, 20, 30]`, give the output of the following Java fragment, or say where and why it fails. Then write the Python equivalent using `iter` and `next` and predict its output.

```
Iterator<Integer> it = list.iterator();
System.out.println(it.next());
System.out.println(it.hasNext());
System.out.println(it.next() + it.next());
System.out.println(it.hasNext());
System.out.println(it.next());
```

Exercise 13.2 (ListIterator). Using a `ListIterator`, write a Java method `void doubleLong(List<String> list)` that inserts a copy of every element longer than three characters right after it, in a single pass, so that `[wash, dry, iron]` becomes `[wash, wash, dry, iron, iron]`. Write the Python equivalent. Why must the Python version not simply loop over the list while inserting?

Exercise 13.3 (an iterable of your own). The laundry picks up and delivers every day except Sunday. Write `PickupDays`, an iterable over the dates from a first to a last date inclusive, skipping Sundays, in Java (implementing `Iterable<LocalDate>`) and in Python (with a generator). Print the days from 25 September to 1 October 2026 with their weekday names, and count the pickup days in September 2026.

Exercise 13.4 (concurrent modification). For each fragment, say whether it throws `ConcurrentModificationException`, silently gives a wrong result, or works, for the list `[1, 2, 3, 4]`; explain with Equation (13.1) and Equation (13.2). (a) `for (Integer x : list) if (x == 3) list.remove(x);` (b) `for (Integer x : list) if (x == 2) list.remove(x);` (c) `for (Integer x : list) if (x == 4) list.add(5);` (d) `for (int i = 0; i < list.size(); i++) if (list.get(i) % 2 == 0) list.remove(i);` Note that `list.remove(x)` with an `Integer` argument removes by value.

Exercise 13.5 (comparators). Write comparators, in Java with `Comparator.comparing`/`thenComparing` and in Python with key functions, that sort the laundry's `Order` objects (a) by total, largest first, ties by id; (b) by customer name, then by date, newest first; (c) members first, then by weight. Then write, for the name data of **Listing 13.11**, a comparator that orders by last name and then first name *ignoring case*.

Exercise 13.6 (the contract). Show that the comparison `(a, b) -> a - b` for `int`s violates Equation (13.3) by giving concrete values, and explain why `Integer.compare(a, b)` does not. Then show that the following comparison of wash codes violates the contract: `compare(a, b)` returns -1 if `a.length() < b.length()`, 1 if `a.charAt(0) > b.charAt(0)`, and 0 otherwise, checked in that order. Find codes among `"WF"`, `"IR"`, `"WDI"`, `"DC"` that demonstrate the violation.

Exercise 13.7 (mutable or immutable?). For each of the following, say whether it would be useful to make it mutable or immutable, and why: (a) a color; (b) a bank account; (c) a 2D coordinate or "point"; (d) the `Integer` class, which holds an `int` value (useful for generics, since generics cannot hold primitive types); (e) a student class that must have a name, a date of birth, an address and an ID number.

Exercise 13.8 (find the leaks). The following class is meant to be immutable. List every way in which it can be changed after construction, and rewrite it correctly in Java (as a record) and in Python (as a frozen dataclass).

```
public class Voucher {
    public final String code;
    private final java.util.Date expires;
    private final List<String> services;
    public Voucher(String code, java.util.Date expires, List<String> services) {
        this.code = code; this.expires = expires; this.services = services;
    }
    public java.util.Date getExpires() { return expires; }
    public List<String> getServices() { return services; }
}
```

Exercise 13.9 (copies). Predict the output of the following Python program, then check it. Draw the objects in the style of **Figure 13.3**.

```
import copy
a = [[1], [2]]
b = a
c = a[:]
d = copy.deepcopy(a)
a[0].append(10)
a.append([3])
print(a, b, c, d)
```

Exercise 13.10 (hashing). Java's `String.hashCode` for `"WF"` is $31 \cdot 87 + 70 = 2767$ and for `"IR"` is $31 \cdot 73 + 82 = 2345$. Using Equation (13.7) with $m = 16$, find the bucket of each. Then explain what would go wrong if `String` were mutable and someone changed the key `"WF"` to `"IR"` after using it in a `HashMap<String, Price>`.

Exercise 13.11 (more questions for the ledger). Without changing `LaundryLedger.java` or `laundry_ledger.py`, write programs that print: (a) the total paid by members and by non-members; (b) the single most expensive order line in the history; (c) the number of orders per day and the busiest day (the earliest if there is a tie); (d) for each customer, the sorted set of service codes he or she has used; (e) the id of the first order whose total exceeds Rp60,000, found lazily.

Exercise 13.12 (laziness). In Java, `Stream.of(1, 2, 3, 4).filter(x -> { System.out.println("f" + x); return x % 2 == 0; }).map(x -> { System.out.println("m" + x); return x * 10; }).findFirst();` prints some lines. Predict them. Then write the same pipeline in Python with a generator expression and predict what `next(...)` prints. What would be printed if the Python version used a list comprehension instead?

Exercise 13.13 (money without losing a rupiah). Three friends share a laundry bill of Rp100,000. `Money.of(100_000 / 3)` is Rp33,333, and three of those make only Rp99,999. Write a function `split(total, k)` that returns k amounts that differ by at most one rupiah and add up exactly to the total, and a function `allocate(total, weights...)` that splits in proportion to integer weights, giving any leftover rupiah to the first shares. Use the `Money` of Section 13.11 without modifying it. Apply `allocate` to the ledger's revenue with the weights 50, 30, 20.

13.16 Solutions to the Exercises

Solution 13.1. The fragment prints `10`, `true`, `50` (20 + 30), `false`, and then the last `it.next()` throws `NoSuchElementException`, because the cursor is at gap 3 of 3 and Equation (13.1) says there is no next element. In Python:

```
it = iter([10, 20, 30])
print(next(it))              # 10
print(next(it) + next(it))   # 50
print(next(it, "done"))      # done   (without the default: StopIteration)
```

Python has no `hasNext`, so the two `hasNext` lines have no direct counterpart; `next(it, "done")` is the idiomatic check.

Solution 13.2. In Java:

```
static void doubleLong(List<String> list) {
    ListIterator<String> it = list.listIterator();
    while (it.hasNext()) {
        String s = it.next();
        if (s.length() > 3) {
            it.add(s);   // inserted just before the cursor, i.e. right after s
        }
    }
}
```

`add` inserts the new element *before* the cursor, which after `next()` means right after `s`, and leaves the cursor after the inserted copy; the next `next()` therefore returns the element that originally followed `s`, and the copy is never visited. Because the change is made through the iterator, there is no `ConcurrentModificationException`. For `[wash, dry, iron]` the result is `[wash, wash, dry, iron, iron]`. In Python, build a new list, `result = [y for x in items for y in ((x, x) if len(x) > 3 else (x,))]`, and if the old list object must be updated, assign `items[:] = result`. Inserting into a list while a `for` loop walks it would make the loop visit each inserted copy — which is also longer than three characters and would be copied again, forever — the problem of Section 13.5 in its most dramatic form.

Solution 13.3. **Listing 13.27** and **Listing 13.28**.

Listing 13.27 `PickupDays.java`: an `Iterable<LocalDate>` that skips Sundays, with an anonymous inner iterator.

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class PickupDays implements Iterable<LocalDate> {
    private final LocalDate first;
    private final LocalDate last;

    public PickupDays(LocalDate first, LocalDate last) {
        if (last.isBefore(first)) {
            throw new IllegalArgumentException("last is before first");
        }
        this.first = first;
        this.last = last;
    }

    private static LocalDate skipSunday(LocalDate day) {
        return day.getDayOfWeek() == DayOfWeek.SUNDAY ? day.plusDays(1) : day;
    }

    @Override
    public Iterator<LocalDate> iterator() {
        return new Iterator<>() {
            private LocalDate next = skipSunday(first);   // LocalDate is immutable

            @Override
            public boolean hasNext() {
                return !next.isAfter(last);
            }

            @Override
            public LocalDate next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                LocalDate result = next;
                next = skipSunday(next.plusDays(1));      // a new date; result is
untouched
                return result;
            }
        };
    }
```

Listing 13.27 (Continued) *PickupDays.java: an Iterable<LocalDate> that skips Sundays, with an anonymous inner iterator.*

```
    public static void main(String[] args) {
        PickupDays days = new PickupDays(LocalDate.of(2026, 9, 25), LocalDate.of(2026, 10,
1));
        for (LocalDate d : days) {
            System.out.println(d + " " + d.getDayOfWeek());
        }
        int n = 0;
        for (LocalDate ignored : new PickupDays(LocalDate.of(2026, 9, 1),
                                                LocalDate.of(2026, 9, 30))) {
            n++;
        }
        System.out.println("pickup days in September 2026: " + n);
    }
}
```

```
2026-09-25 FRIDAY
2026-09-26 SATURDAY
2026-09-28 MONDAY
2026-09-29 TUESDAY
2026-09-30 WEDNESDAY
2026-10-01 THURSDAY
pickup days in September 2026: 26
```

Listing 13.28 `pickup_days.py`: the same iterable as a generator.

```
"""pickup_days.py: an iterable of dates that skips Sundays."""
from datetime import date, timedelta


class PickupDays:
    def __init__(self, first, last):
        if last < first:
            raise ValueError("last is before first")
        self.first = first
        self.last = last

    def __iter__(self):
        day = self.first
        while day <= self.last:
            if day.weekday() != 6:          # Monday is 0, Sunday is 6
                yield day
            day += timedelta(days=1)        # a new date object; the old one is untouched


def main():
    for d in PickupDays(date(2026, 9, 25), date(2026, 10, 1)):
        print(d, d.strftime("%A").upper())
    n = sum(1 for _ in PickupDays(date(2026, 9, 1), date(2026, 9, 30)))
    print("pickup days in September 2026:", n)


if __name__ == "__main__":
    main()
```

```
2026-09-25 FRIDAY
2026-09-26 SATURDAY
2026-09-28 MONDAY
2026-09-29 TUESDAY
2026-09-30 WEDNESDAY
2026-10-01 THURSDAY
pickup days in September 2026: 26
```

The Java iterator stores the next date to return, already moved past any Sunday; `LocalDate` is immutable, so `next.plusDays(1)` creates a new date and the returned `result` can never be changed by the iterator later — a small but real benefit of immutability. September 2026 has 30 days, four of which (the 6th, 13th, 20th and 27th) are Sundays, leaving 26 pickup days. The Python generator is simpler still: loop over the days and `yield` those that are not Sundays (`weekday()` is 6 for Sunday).

Solution 13.4. (a) Removing 3, the *second-to-last* element: after `next()` returned 3, the cursor $c = 3$; the list shrinks to $n = 3$; `hasNext()` is $3 \neq 3$, false; the loop ends without checking Equation (13.2), and 4 is never visited. It *works by accident* — no exception, and in this case the result `[1, 2, 4]` is even correct, which makes the bug harder to notice. (b) Removing 2: $c = 2$, n becomes 3, `hasNext()` is true, and `next()` finds `modCount` $\neq$ `expectedModCount`: `ConcurrentModificationException`. (c) Adding after the last element: after `next()` returned 4, $c = 4$, n becomes 5, `hasNext()` is true, `next()` checks Equation (13.2): exception. (d) The index

loop has no iterator, so nothing throws, but it skips: at $i = 1$ it removes 2, 3 moves to index 1, i becomes 2 and looks at 4, which is removed; the result is `[1, 3]`, correct by luck. With `[2, 4, 6]` it would give `[4]`. Iterating the index downwards is correct (Section 13.3.2); `list.removeIf(x -> x % 2 == 0)` is simplest.

Solution 13.5. In Java, with the records of **Listing 13.25**:

```
Comparator<Order> a = Comparator.comparing(Order::total).reversed()
        .thenComparing(Order::id);
Comparator<Order> b = Comparator.comparing(Order::customer)
        .thenComparing(Order::date, Comparator.reverseOrder());
Comparator<Order> c = Comparator.comparing((Order o) -> !o.member())
        .thenComparingInt(Order::weightTenths);
Comparator<Name> ci = Comparator.comparing(Name::lastName, String.CASE_INSENSITIVE_ORDER)
        .thenComparing(Name::firstName, String.CASE_INSENSITIVE_ORDER);
```

In (c), `false` sorts before `true`, so `!member` puts members first. In Python: (a) `key=lambda o: (-o.total().rupiah, o.id)`; (b) sort twice, relying on stability: first `sort(key=attrgetter("date"), reverse=True)`, then `sort(key=attrgetter("customer"))` — or use `key=lambda o: (o.customer, -o.date.toordinal())`; (c) `key=lambda o: (not o.member, o.weight_tenths())`; the case-insensitive name order is `key=lambda n: (n.last_name.casefold(), n.first_name.casefold())`. Negating a number reverses one part of a tuple key; for strings, use the two-pass stable sort instead.

Solution 13.6. Take $a = 2^{31} - 1$ (`Integer.MAX_VALUE`) and $b = -1$. Then $a - b$ overflows to -2^{31} (Equation (13.5)), so `cmp(a, b) < 0`; and $b - a = -2^{31}$, which is representable, so `cmp(b, a) < 0` as well. Both say "less", violating Equation (13.3). `Integer.compare(a, b)` is implemented as `(a < b) ? -1 : ((a == b) ? 0 : 1)`, which compares instead of subtracting and cannot overflow.

For the wash codes, work out both directions for one pair. `compare("WF", "IR")`: the lengths are equal, so the first rule does not apply; 'W' > 'I', so the result is 1. `compare("IR", "WF")`: equal lengths again; 'I' is not greater than 'W', so the result is 0. Antisymmetry demands -1, so Equation (13.3) is violated. The third rule of the contract fails too: `compare("WDI", "WF")` is 0 ("equal"), so `"WDI"` and `"WF"` should compare the same way with every other code; yet `compare("IR", "WDI")` is -1 (shorter) while `compare("IR", "WF")` is 0. A comparison assembled from ad hoc rules checked "in some order" almost never forms a valid ordering; build comparisons from keys, field by field.

Solution 13.7. (a) **Color: immutable.** A color is a value; "changing" red to blue means using a different color, and colors are shared everywhere (Java's `java.awt.Color` is immutable). (b) **Bank account: mutable**, or rather an *entity*: its balance must change over time, and it has an identity — this account. Its *transactions*, however, are best made immutable values, and the balance can be derived from them, like the laundry's `OrderHistory`. (c) **Point: immutable.** It is a value, and points are natural keys (a map from positions to machines). (d) **`Integer`: immutable** — and it is. `Integer` objects are cached and shared (for small values, `Integer.valueOf(7)` returns the same object every time); if one could be changed, every 7 in the program would change with it. (e) **Student: mostly immutable.** Name, date of birth and ID identify the student and should be `final`; the ID must never change because it is the key in every map of students. An address does change, so either the class offers `withAddress`, returning a new student, or the address is kept in a separate map from student ID to address. The fields must themselves be primitives or immutable (`int`, `String`, `LocalDate` — not `java.util.Date`).

Solution 13.8. Leaks: (1) the constructor keeps the caller's `Date`, which the caller can change with `setTime`; (2) `getExpires()` returns that same mutable `Date`; (3) the constructor keeps the caller's

list; (4) `getServices()` returns the internal list; (5) the class is not `final`, so a subclass could add mutable state or override the getters. (`code` is `public final` and a `String`, which is safe, if unusual style.) Correct versions:

```
public record Voucher(String code, LocalDate expires, List<String> services) {
    public Voucher {
        Objects.requireNonNull(code);
        Objects.requireNonNull(expires);          // LocalDate is immutable: no copy
needed
        services = List.copyOf(services);         // defensive, unmodifiable copy
    }
}
```

```
@dataclass(frozen=True)
class Voucher:
    code: str
    expires: date
    services: tuple
    def __post_init__(self):
        object.__setattr__(self, "services", tuple(self.services))
```

Replacing `java.util.Date` (mutable) with `java.time.LocalDate` (immutable) removes two leaks at once; if you must keep `Date`, copy it both ways with `new Date(d.getTime())`.

Solution 13.9. The program prints `[[1, 10], [2], [3]] [[1, 10], [2], [3]] [[1, 10], [2]] [[1], [2]]`. `b` is an alias of `a`, so it shows both changes. `c` is a shallow copy: a different outer list (so it lacks `[3]`) holding the same inner lists (so it shows the 10). `d` is a deep copy and shows neither change.

Solution 13.10. $2767 = 172 \cdot 16 + 15$, so `"WF"` goes to bucket 15; $2345 = 146 \cdot 16 + 9$, so `"IR"` goes to bucket 9. (Java also mixes in $h \gg 16$, which is 0 for these small codes.) If the stored key object `"WF"` could be changed into `"IR"`, the entry would stay in bucket 15 while every lookup of `"IR"` went to bucket 9 and found nothing, and a lookup of `"WF"` would reach bucket 15 but fail the `equals` test — the situation of **Figure 13.4**. Worse, a later `put("IR", ...)` would create a second entry with an equal key. This is one of the reasons `String` was made immutable.

Solution 13.11. **Listing 13.29** and **Listing 13.30** answer all five questions using only the public parts of the ledger.

Listing 13.29 `LedgerQueries.java`: five further questions answered with streams, including `partitioningBy`, `flatMap`, `max`, `counting` and `flatMapping`.

```
import java.time.LocalDate;
import java.util.Comparator;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import java.util.TreeSet;
import java.util.stream.Collectors;

// Needs LaundryLedger.java in the same folder (compile both with javac *.java).
public class LedgerQueries {
    public static void main(String[] args) {
        OrderHistory history = LaundryLedger.septemberHistory();

        // (a) Members versus non-members: partitioningBy splits a stream in two.
        Map<Boolean, Money> byMembership = history.stream()
                .collect(Collectors.partitioningBy(Order::member,
                        Collectors.reducing(Money.ZERO, Order::total, Money::plus)));
        System.out.println("members " + byMembership.get(true)
                + ", others " + byMembership.get(false));

        // (b) The single most expensive line in the whole history.
        OrderLine biggest = history.stream()
                .flatMap(o -> o.lines().stream())
                .max(Comparator.comparing(OrderLine::amount))
                .orElseThrow();
        System.out.println("biggest line: " + biggest.price().name() + " "
                + biggest.weightTenths() / 10.0 + " kg = " + biggest.amount());

        // (c) Number of orders per day, and the busiest day.
        Map<LocalDate, Long> ordersPerDay = history.stream()
                .collect(Collectors.groupingBy(Order::date, TreeMap::new,
                        Collectors.counting()));
        LocalDate busiest = ordersPerDay.entrySet().stream()
                .max(Map.Entry.<LocalDate, Long>comparingByValue()

.thenComparing(Map.Entry.comparingByKey(Comparator.reverseOrder())))
                .orElseThrow().getKey();
        System.out.println("orders per day " + ordersPerDay + ", busiest " + busiest);

        // (d) The set of services each customer has used, sorted by name.
        Map<String, Set<String>> services = history.stream()
                .collect(Collectors.groupingBy(Order::customer, TreeMap::new,
                        Collectors.flatMapping(o -> o.lines().stream().map(l ->
l.price().code()),
                                Collectors.toCollection(TreeSet::new))));
        services.forEach((name, codes) -> System.out.println("  " + name + " used " +
codes));
```

Listing 13.29 (Continued) *LedgerQueries.java: five further questions answered with streams, including partitioningBy, flatMap, max, counting and flatMapping.*

```
        // (e) The first order whose total exceeds Rp60,000, found lazily.
        String firstBig = history.stream()
                .filter(o -> o.total().compareTo(Money.of(60_000)) > 0)
                .map(Order::id)
                .findFirst()
                .orElse("none");
        System.out.println("first order over Rp60,000: " + firstBig
                + " (of " + history.size() + ")");
    }
}
```

```
members Rp106,650, others Rp223,000
biggest line: Wash & Fold 7.9 kg = Rp56,000
orders per day {2026-09-21=3, 2026-09-22=3, 2026-09-23=2}, busiest 2026-09-21
  Ayu Lestari used [WDI, WF]
  Budi Santoso used [IR, WDI]
  Dewi Anggraini used [IR, WF]
  Siti Aminah used [IR, WDI, WF]
first order over Rp60,000: ORD-1305 (of 8)
```

Listing 13.30 `ledger_queries.py`: the same five questions with comprehensions, `max` with a key, `Counter` and `next` with a default.

```
"""ledger_queries.py: more questions for the laundry ledger (needs laundry_ledger.py)."""
from collections import Counter

from laundry_ledger import ZERO, september_history


def main():
    history = september_history()
    orders = list(history.oldest_first())

    # (a) Members versus non-members.
    members = sum((o.total() for o in orders if o.member), ZERO)
    others = sum((o.total() for o in orders if not o.member), ZERO)
    print(f"members {members}, others {others}")

    # (b) The single most expensive line in the whole history.
    biggest = max((line for o in orders for line in o.lines), key=lambda ln: ln.amount())
    print(f"biggest line: {biggest.price.name} {biggest.weight_tenths / 10} kg = "
          f"{biggest.amount()}")

    # (c) Number of orders per day, and the busiest day (earliest wins a tie).
    per_day = Counter(o.date for o in orders)
    busiest = min(per_day, key=lambda d: (-per_day[d], d))
    print("orders per day", {str(d): n for d, n in sorted(per_day.items())},
          "busiest", busiest)

    # (d) The set of services each customer has used, sorted by name.
    services = {name: sorted({ln.price.code for o in orders if o.customer == name
                              for ln in o.lines})
                for name in sorted({o.customer for o in orders})}
    for name, codes in services.items():
        print(f"  {name} used {codes}")

    # (e) The first order whose total exceeds Rp60,000, found lazily.
    first_big = next((o.id for o in orders if o.total().rupiah > 60_000), "none")
    print(f"first order over Rp60,000: {first_big} (of {len(orders)})")


if __name__ == "__main__":
    main()
```

```
members Rp106,650, others Rp223,000
biggest line: Wash & Fold 7.9 kg = Rp56,000
orders per day {'2026-09-21': 3, '2026-09-22': 3, '2026-09-23': 2} busiest 2026-09-21
  Ayu Lestari used ['WDI', 'WF']
  Budi Santoso used ['IR', 'WDI']
  Dewi Anggraini used ['IR', 'WF']
  Siti Aminah used ['IR', 'WDI', 'WF']
first order over Rp60,000: ORD-1305 (of 8)
```

(a) `partitioningBy` is a `groupingBy` with a `boolean` key; Rp106,650 plus Rp223,000 is the revenue of Rp329,650. (b) The largest line is the 7.9 kg Wash & Fold of ORD-1305: $\lceil 79/5 \rceil = 16$, so 80 tenths and $7000 \cdot 80/10 = 56{,}000$. (c) To prefer the earliest tied day, Java breaks

ties by *reversed* date inside `max`; Python minimizes `(-count, date)`. (d) `flatMapping` (Java 9+) collects each customer's codes into a `TreeSet`; Python nests two `for` clauses. (e) `findFirst` and `next(..., "none")` stop at ORD-1305.

Solution 13.12. Java streams pass one element at a time through the whole pipeline, so the output is `f1`, `f2`, `m2`, and then `findFirst` has its answer and the stream stops: 1 fails the filter, 2 passes and is mapped, and 3 and 4 are never touched. (If you predicted `f1 f2 f3 f4 m2 m4`, you were imagining each operation running over the whole list before the next one starts, which is how eager lists behave.) In Python, with helper functions `f(x)` and `m(x)` that print and return, `next(m(x) for x in [1, 2, 3, 4] if f(x))` prints the same `f1 f2 m2`, because generator expressions are lazy too. The list comprehension `[m(x) for x in [1, 2, 3, 4] if f(x)][0]` runs to the end before the first element can be taken, and prints `f1 f2 m2 f3 f4 m4`.

Solution 13.13. Listing 13.31 and **Listing 13.32**. Dividing a total T into k parts, let

$$q = \left\lfloor \frac{T}{k} \right\rfloor, \qquad r = T - kq, \qquad \text{parts} = \underbrace{q+1, \ldots, q+1}_{r}, \underbrace{q, \ldots, q}_{k-r} \tag{13.10}$$

Then the parts sum to $r(q+1) + (k-r)q = kq + r = T$ exactly, and any two differ by at most 1 (Equation (13.10)).

Listing 13.31 `MoneySplit.java`: splitting and allocating `Money` so that the parts always add up to the total.

```
import java.util.ArrayList;
import java.util.List;

// Needs LaundryLedger.java in the same folder (it defines the Money record).
public class MoneySplit {
    // k parts that differ by at most one rupiah and add up exactly to the total.
    static List<Money> split(Money total, int k) {
        if (k <= 0) {
            throw new IllegalArgumentException("k must be positive");
        }
        long q = total.rupiah() / k;
        long r = total.rupiah() % k;            // r parts get one rupiah more
        List<Money> parts = new ArrayList<>();
        for (int i = 0; i < k; i++) {
            parts.add(Money.of(i < r ? q + 1 : q));
        }
        return List.copyOf(parts);
    }

    // Shares proportional to the weights; leftover rupiah go to the first shares.
    static List<Money> allocate(Money total, int... weights) {
        long sum = 0;
        for (int w : weights) {
            sum += w;
        }
        List<Money> parts = new ArrayList<>();
        long given = 0;
        for (int w : weights) {
            long share = total.rupiah() * w / sum;
            parts.add(Money.of(share));
            given += share;
        }
        for (int i = 0; given < total.rupiah(); i++, given++) {
            parts.set(i, parts.get(i).plus(Money.of(1)));
        }
        return List.copyOf(parts);
    }

    static Money sum(List<Money> parts) {
        return parts.stream().reduce(Money.ZERO, Money::plus);
    }
```

Listing 13.31 (Continued) *MoneySplit.java: splitting and allocating Money so that the parts always add up to the total.*

```
    public static void main(String[] args) {
        Money bill = Money.of(100_000);
        List<Money> three = split(bill, 3);
        System.out.println("split in 3:     " + three + " sum " + sum(three));
        System.out.println("naive third x3: " + Money.of(bill.rupiah() / 3).times(3));
        List<Money> shares = allocate(Money.of(329_650), 50, 30, 20);
        System.out.println("50/30/20:       " + shares + " sum " + sum(shares));
        List<Money> odd = allocate(Money.of(10), 1, 1, 1);
        System.out.println("Rp10 by 1:1:1:  " + odd + " sum " + sum(odd));
    }
}
```

```
split in 3:     [Rp33,334, Rp33,333, Rp33,333] sum Rp100,000
naive third x3: Rp99,999
50/30/20:       [Rp164,825, Rp98,895, Rp65,930] sum Rp329,650
Rp10 by 1:1:1:  [Rp4, Rp3, Rp3] sum Rp10
```

Listing 13.32 `money_split.py`: the same with `divmod` and a tuple of `Money` values.

```
"""money_split.py: splitting an amount without losing a rupiah (needs
laundry_ledger.py)."""
from laundry_ledger import ZERO, Money


def split(total, k):
    """k parts that differ by at most one rupiah and add up exactly to the total."""
    if k <= 0:
        raise ValueError("k must be positive")
    q, r = divmod(total.rupiah, k)            # r parts get one rupiah more
    return tuple(Money(q + 1 if i < r else q) for i in range(k))


def allocate(total, *weights):
    """Shares proportional to the weights; leftover rupiah go to the first shares."""
    s = sum(weights)
    shares = [total.rupiah * w // s for w in weights]
    for i in range(total.rupiah - sum(shares)):
        shares[i] += 1
    return tuple(Money(x) for x in shares)


def show(parts):
    return "[" + ", ".join(str(p) for p in parts) + "]"


def main():
    bill = Money(100_000)
    three = split(bill, 3)
    print("split in 3:    ", show(three), "sum", sum(three, ZERO))
    print("naive third x3:", Money(bill.rupiah // 3) * 3)
    shares = allocate(Money(329_650), 50, 30, 20)
    print("50/30/20:      ", show(shares), "sum", sum(shares, ZERO))
    odd = allocate(Money(10), 1, 1, 1)
    print("Rp10 by 1:1:1: ", show(odd), "sum", sum(odd, ZERO))


if __name__ == "__main__":
    main()
```

```
split in 3:     [Rp33,334, Rp33,333, Rp33,333] sum Rp100,000
naive third x3: Rp99,999
50/30/20:       [Rp164,825, Rp98,895, Rp65,930] sum Rp329,650
Rp10 by 1:1:1:  [Rp4, Rp3, Rp3] sum Rp10
```

For Rp100,000 in 3, $q = 33{,}333$ and $r = 1$, so the parts are Rp33,334, Rp33,333, Rp33,333. `allocate` first gives each share its rounded-down proportion; the shares then fall short of the total by fewer rupiah than there are shares, and the loop hands those out one at a time. For 50/30/20 of Rp329,650 the proportions are exact, so no leftover is needed; for Rp10 in three equal parts, one leftover rupiah goes to the first share. The results are immutable (`List.copyOf`, `tuple`), and `Money` needed no change at all: an immutable value type can be extended with new *functions* forever without any risk to existing code. Fowler calls this operation *allocation* and recommends exactly this treatment of remainders [19].

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

[2] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Boston, MA, USA: Addison-Wesley, 2000.

[3] Oracle, *Java Platform, Standard Edition 21 API Specification*, packages `java.util` and `java.util.stream`. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/

[4] Python Software Foundation, *The Python Language Reference* and *The Python Standard Library*, version 3.14, sections "Iterator Types", `itertools`, `functools`, `copy` and `dataclasses`. [Online]. Available: https://docs.python.org/3/

[5] K.-P. Yee and G. van Rossum, "PEP 234 – Iterators," Python Enhancement Proposal 234, 2001. [Online]. Available: https://peps.python.org/pep-0234/

[6] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*. Oracle America, 2023. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se21/html/

[7] N. Schemenauer, T. Peters, and M. L. Hetland, "PEP 255 – Simple generators," Python Enhancement Proposal 255, 2001. [Online]. Available: https://peps.python.org/pep-0255/

[8] L. Ramalho, *Fluent Python*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2022.

[9] J. Bloch, *Effective Java*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2018.

[10] T. Peters, "listsort.txt: description of the list sort algorithm (Timsort)," CPython source distribution, 2002. [Online]. Available: https://github.com/python/cpython/blob/main/Objects/listsort.txt

[11] Oracle, "Immutable objects," *The Java Tutorials*, Concurrency lesson. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

[12] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Upper Saddle River, NJ, USA: Addison-Wesley, 2006.

[13] S. Marks, "JEP 269: Convenience factory methods for collections," OpenJDK, 2017. [Online]. Available: https://openjdk.org/jeps/269

[14] G. Bierman, "JEP 395: Records," OpenJDK, 2020. [Online]. Available: https://openjdk.org/jeps/395

[15] E. V. Smith, "PEP 557 – Data classes," Python Enhancement Proposal 557, 2017. [Online]. Available: https://peps.python.org/pep-0557/

[16] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA, USA: Addison-Wesley, 2003.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[18] R. Warburton, *Java 8 Lambdas: Functional Programming for the Masses*. Sebastopol, CA, USA: O'Reilly Media, 2014.

[19] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2002.

Chapter 14

Threads, Races, Deadlock, and Livelock

By the end of this chapter you will be able to explain what a thread is and how it differs from a program, and say when running several threads makes a program faster, more responsive or simply easier to write; create threads in Java with `Thread`, `Runnable` and an `ExecutorService`, and in Python with `threading.Thread` and `concurrent.futures`; stop a thread politely with an interrupt, a `volatile` flag or an `Event`; follow a thread through its life, from `NEW` to `TERMINATED`; explain what Python's Global Interpreter Lock is and predict when Python threads help and when they do not; reproduce a *race condition* on a shared counter, read the machine instructions that cause it, and cure it with `synchronized`, a `ReentrantLock`, an `AtomicInteger` or a `threading.Lock`; make threads wait for each other with `wait`/`notifyAll` and `Condition`, and then let a `BlockingQueue` or `queue.Queue` do the waiting for you; recognize a *deadlock* in the dining philosophers and in two clerks who each hold one machine, prove that it is one with a wait-for graph, and prevent it by lock ordering, by `tryLock` with a timeout, or by an arbitrator; recognize the quieter failures called *livelock* and *starvation*; and build a small laundry in which three washing machines, each a thread, take orders from one queue and pay one cashier without ever losing a rupiah. Throughout, we are honest about a property that makes this subject different from everything before it: a concurrent program can print something different every time you run it, and we will show real runs and say exactly what varies and what must never vary.

14.1 Where We Left Off

Every program in this book so far has done one thing at a time. When `main` called a method, `main` stopped and waited until the method returned; when a loop ran, nothing else happened until the loop ended. Chapter 10 was the one quiet exception. When you clicked a button in a Swing or Tkinter window, your handler ran, and yet the window kept repainting itself, and you never wrote the loop that watched the mouse. Somebody else was running that loop for you. In Swing it is a separate *thread*, the Event Dispatch Thread; in Tkinter it is the `mainloop`. Chapter 11 split the program into model, view and controller, and Chapter 13 made objects immutable, a property that will turn out to be one of the best defenses we have in this chapter.

Now suppose that Profitina Laundry buys two more washing machines. Three machines can wash three loads at the same moment. The shop's software, however, still does one thing at a time: it starts a wash, waits for it, records the payment, and only then starts the next. Two of the three machines stand idle most of the day. What the shop needs is software in which each machine has its own little program running *at the same time* as the others, all of them reading from one list of waiting orders and all of them paying into one cash register.

That is what threads give us, and it is also where the trouble starts. Two machines that take the same order, or two payments that are added to the cash register at the same instant and leave only one of them in the total, are not hypothetical bugs. We will produce both of them on purpose, on real hardware, and then learn the disciplines that prevent them.

✓ **KEY INSIGHT.** Threads are easy to *start* and hard to *coordinate*. Almost everything that goes wrong in a concurrent program comes from one situation: two threads touch the same data, and at least one of them changes it. Every tool in this chapter (locks, atomic variables, conditions, queues, lock ordering) exists to control that one situation.

14.2 What Is a Thread?

Let us begin with the shop itself, because the laundry is a very good model of a computer. Imagine Profitina Laundry on a busy Saturday. There are three washing machines. Each machine does its own work: it takes a bag of laundry, washes it, and when it finishes, somebody takes the receipt to the cashier. There is only **one** cashier, and the cashier can serve only one person at a time. Customers bring orders to a front desk, and the orders wait in a basket until a machine is free.

Much of this is *simultaneous*: all three machines wash at once, and the front desk accepts new orders meanwhile. But some things must happen *one at a time*: the cashier writes one payment in the book, then the next; two people writing in the same line at once would make the book unreadable. **Figure 14.1** draws the shop exactly this way, and it is also, almost word for word, the design of the program we will build in Section 14.14.

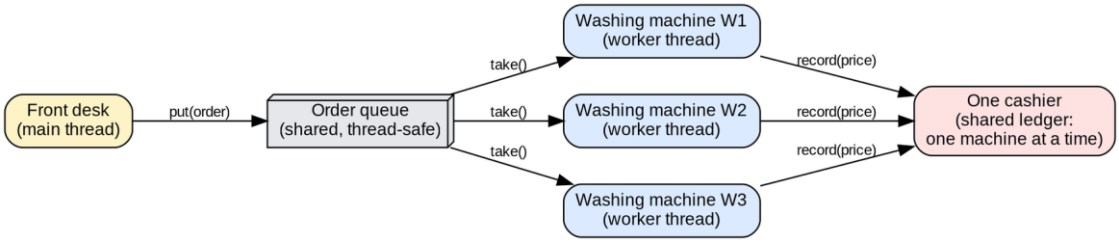


Figure 14.1 Profitina Laundry as a concurrent system. The front desk puts orders into one shared queue; three washing machines, each a worker thread, take orders from it; all three pay into one ledger, which accepts one machine at a time.

Now translate the picture into computing words.

DEFINITION. A **process** is a running program: the operating system gives it its own memory, its own open files and at least one thread. A **thread** is one independent sequence of instructions running inside a process. Every thread has its own *call stack* (its own local variables and its own record of which method called which), but all the threads of a process share the same *heap*: the same objects, the same fields, the same lists.

So each washing machine is a thread with its own work in hand (its local variables: which order, how many kilograms). The basket of orders and the cashier's book are shared objects on the heap that every machine can see and change. That sharing makes threads useful (the machines cooperate through the basket) and dangerous (they can spoil the book by writing at the same time).

A program always has at least one thread. In Java it is called `main`; in Python, `MainThread`. Every program you have written so far ran entirely in that one thread. In this chapter we will create more.

14.2.1 Concurrency and parallelism

Two words are often mixed up, and it is worth separating them now. **Concurrency** means that several tasks are *in progress* at the same time: each has started and none has finished.

Parallelism means that several tasks are literally *executing* at the same instant, on different processor cores.

A computer with one core can run many threads concurrently but never in parallel. The operating system's **scheduler** gives the core to one thread for a short *time slice* (a few milliseconds), then stops it [1], saves its place, and gives the core to the next thread. Because the slices are short, it looks as if all the threads run at once, just as a juggler seems to hold three balls although only one ball is ever in a hand at a given instant. A computer with two or four or sixteen cores can, in addition, run that many threads truly in parallel. The top half of **Figure 14.2** shows three machine threads sharing one core; the bottom half shows the same three threads on two cores, where two of them really do run at the same time and the third waits its turn.

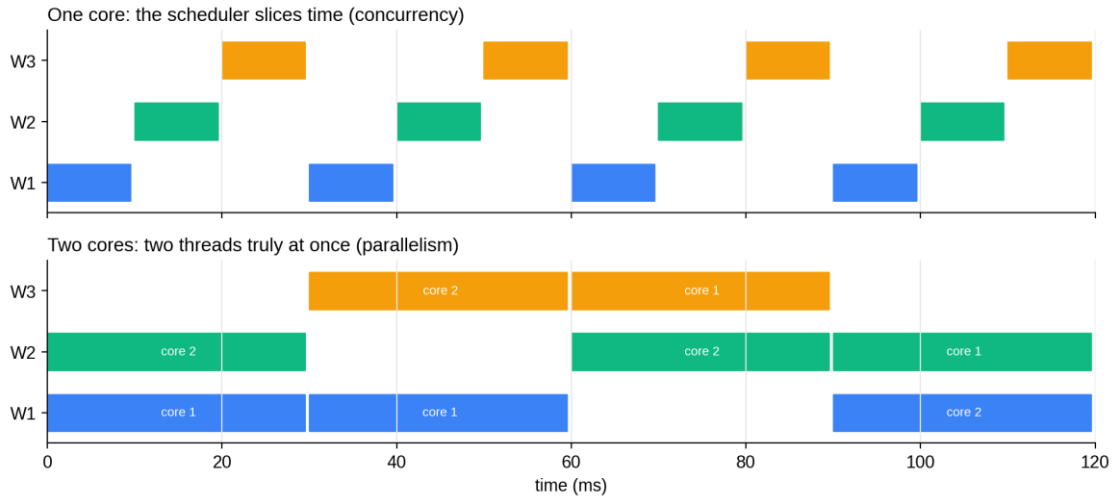


Figure 14.2 Concurrency versus parallelism. Top: one core; the scheduler gives each thread a short time slice in turn, so all three make progress but only one runs at any instant. Bottom: two cores; two threads really run at once, and the scheduler moves threads between cores as it likes.

Two consequences follow from **Figure 14.2**, and the whole chapter rests on them.

- **You do not decide the order.** The scheduler decides when a thread runs, for how long, and when it is paused, and it may decide differently every time you run the program. Code in one thread runs in the order you wrote it; code in *different* threads can interleave in any order at all.
- **A thread can be paused anywhere.** Not only between two statements of your program, but in the middle of what looks like one statement, such as `count++`. Section 14.7 shows why this matters so much.

14.2.2 Why use threads at all?

If threads are this unpredictable, why use them? There are four good reasons [2].

1. **Responsiveness.** A window that computes a report in its only thread freezes until the report is done; one that hands the report to a second thread keeps responding to clicks (Chapter 10).
2. **Waiting in parallel.** While one thread waits for a disk, a network or a washing machine, another can work. A server gives each client a thread (Chapter 15 does exactly this) so that one slow client does not hold up the others.
3. **Using every core.** A laptop today has eight or more cores. A program with one thread uses one of them. A computation that can be split into independent parts can finish several times faster if each part has its own thread and its own core.

4. **Simpler models.** Some problems are naturally several independent activities: three washing machines, five philosophers, a producer and a consumer. Writing each activity as its own thread, with its own simple loop, can be much clearer than one big loop that juggles all of them.

The cost is the subject of the rest of the chapter: when threads share data, we must coordinate them, and coordination is where the bugs live.

14.3 A Parent Who Cannot Get Back Control

Our first example is a family. A parent wants several children to count, each at their own pace, for as long as the parent likes, and then to stop. Let us first try to write it without threads, to see the problem that threads solve. A child has a name and a method `begin` that counts forever:

```
public class Child {
    private String name;
    public Child(String name) { this.name = name; }
    public void begin() {
        int i = 0;
        while (true) {
            System.out.println(name + " has counted " + (i++));
        }
    }
}
```

and the parent creates two children and asks each to begin:

```
Child one = new Child("Luffy");
one.begin();
Child two = new Child("Ace");
two.begin();
```

What happens when this runs? The parent calls `one.begin()`, and control passes to the first child, which enters an infinite loop. The call never returns. The parent never reaches the line that creates the second child, and the parent has no way to stop the first one either: the parent is not running at all, because the parent's only thread is busy inside the child's loop. (We print this program as text rather than as a listing because it never ends; you may type it in and stop it with Ctrl+C.)

What we want is for `begin` to start the counting *somewhere else* and give control back to the parent at once. That is exactly what a thread does.

14.3.1 The children as threads

In Java, a piece of work that can run in its own thread is described by the interface `Runnable`, which has one method, `run()`. We never call `run()` ourselves. Instead we give the `Runnable` to a `Thread` object and call the thread's `start()` method. `start()` asks the Java Virtual Machine to create a new thread, and that new thread calls `run()`; meanwhile `start()` itself returns immediately, and the parent carries on. **Listing 14.1** is the whole program.

Listing 14.1 `ParentChild.java`: a parent starts two counting children as threads, sleeps for three seconds, then interrupts them and waits for them to end.

```
import java.util.Random;

public class ParentChild {
    // A child counts forever, sleeping a random time after every number.
    static class Child implements Runnable {
        private final String name;
        private final Random random;

        Child(String name, long seed) {
            this.name = name;
            this.random = new Random(seed);  // a seed makes the sleep times repeatable
        }

        @Override
        public void run() {
            try {
                int i = 0;
                while (true) {
                    int rand = random.nextInt(1000);
                    System.out.println(name + " has counted to " + (i++)
                            + " and will now sleep for " + rand + "ms");
                    Thread.sleep(rand);
                }
            } catch (InterruptedException e) {
                System.out.println(name + " interrupted; ending");
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        System.out.println("Parent started");
        System.out.println("Parent is starting Child: Luffy");
        Thread threadOne = new Thread(new Child("Luffy", 1));
        threadOne.start();  // start() returns at once; run() goes on in the new thread
        System.out.println("Parent is starting Child: Ace");
        Thread threadTwo = new Thread(new Child("Ace", 2));
        threadTwo.start();
        System.out.println("Parent will sleep for 3 seconds");
        try {
            Thread.sleep(3000);
            System.out.println("Parent has woken up");
        } finally {
            // Ask both children to stop; each decides how to react.
            threadOne.interrupt();
            threadTwo.interrupt();
        }
```

Listing 14.1 (Continued) *ParentChild.java: a parent starts two counting children as threads, sleeps for three seconds, then interrupts them and waits for them to end.*

```
        threadOne.join();  // wait until each child has really finished
        threadTwo.join();
        System.out.println("Parent ended");
    }
}
```

Compile and run it in the chapter's `java` folder with `javac ParentChild.java` and `java ParentChild`. One real run printed:

```
Parent started
Parent is starting Child: Luffy
Parent is starting Child: Ace
Parent will sleep for 3 seconds
Luffy has counted to 0 and will now sleep for 985ms
Ace has counted to 0 and will now sleep for 108ms
Ace has counted to 1 and will now sleep for 372ms
Ace has counted to 2 and will now sleep for 40ms
Ace has counted to 3 and will now sleep for 67ms
Ace has counted to 4 and will now sleep for 389ms
Ace has counted to 5 and will now sleep for 350ms
Luffy has counted to 1 and will now sleep for 588ms
Ace has counted to 6 and will now sleep for 606ms
Luffy has counted to 2 and will now sleep for 847ms
Ace has counted to 7 and will now sleep for 719ms
Luffy has counted to 3 and will now sleep for 313ms
Ace has counted to 8 and will now sleep for 847ms
Luffy has counted to 4 and will now sleep for 254ms
Parent has woken up
Ace interrupted; ending
Luffy interrupted; ending
Parent ended
```

Read the output slowly, because it already contains the essence of concurrency.

- The parent prints `Parent is starting Child: Ace` *immediately* after starting Luffy. `threadOne.start()` did not wait for Luffy's loop; it returned at once, exactly as we wanted.
- The lines of Luffy and Ace are mixed together. Each child prints a line, sleeps for a random time, and prints again, and the two children's sleeps are independent, so their lines interleave. Ace's first sleeps were short (108 ms, 372 ms, 40 ms, 67 ms), so Ace counted to 5 before Luffy woke from his first 985 ms sleep.
- After three seconds the parent wakes, interrupts both children, and each child prints `interrupted; ending`. Only then does the parent print `Parent ended`, because `join()` waits until a thread has finished.

Now the line-by-line reading.

- `static class Child implements Runnable` declares the work. A *static nested class* is simply a class written inside another class, which keeps the whole example in one file; it behaves like any top-level class.
- `this.random = new Random(seed)` gives each child its own random number generator with a fixed *seed*, so the sleep times (985, 588, 847, … for Luffy) are the same on every run and on your computer.
- `while (true) { … Thread.sleep(rand); }` is the child's work: print, then sleep for `rand` milliseconds. `Thread.sleep` pauses *the thread that calls it*, here the child, and lets other threads run.
- `catch (InterruptedException e)` is the child's reaction to being interrupted. `Thread.sleep` is one of the methods that throws `InterruptedException` when another thread calls `interrupt()` on the sleeping thread. The exception jumps out of the loop, the child prints a farewell, `run()` returns, and the thread ends.
- In `main`, `new Thread(new Child("Luffy", 1))` wraps the work in a thread, and `threadOne.start()` sets it going. The parent continues at once.
- `Thread.sleep(3000)` puts *the parent*, which is simply the `main` thread, to sleep for three seconds.
- The `finally` block calls `interrupt()` on both children. It sits in `finally` so that the children are asked to stop even if the parent's sleep were somehow cut short.

- `threadOne.join()` blocks the parent until Luffy's thread has ended. Without the two `join` calls, `Parent ended` might be printed before the children's farewells.

What *varies* between runs? Not the sleep times, which the seeds fix, but the fine timing: two wake-ups due within a millisecond of each other may print in either order, a child whose wake-up falls near 3,000 ms may print one line more or less, and the two `interrupted; ending` lines come in either order. The pattern is the same on every run; the details are not.

> ⚠ **WARNING.** An interrupt is a *request*, not a command. `interrupt()` only sets a flag in the target thread and wakes it if it is sleeping or waiting. It is entirely the child's choice what to do: our children end, but a child that caught `InterruptedException` and simply looped again would ignore the parent forever. Java has no safe way to kill a thread from outside; the old `Thread.stop()` was deprecated because a thread killed in the middle of an update leaves objects half-changed [3].

Listing 14.2 is the same family in Python. The module `threading` [4] provides `threading.Thread`, which takes the work as a *function* (the `target`) rather than as an object implementing an interface.

Listing 14.2 `parent_child.py`: the same parent and children in Python; a shared `threading.Event` replaces Java's interrupt.

```
"""parent_child.py: a parent starts two counting children and later stops them."""
import random
import threading


class Child:
    """A child counts forever, sleeping a random time after every number."""

    def __init__(self, name, seed, stop):
        self.name = name
        self.random = random.Random(seed)  # a seed makes the sleep times repeatable
        self.stop = stop                   # a threading.Event shared with the parent

    def run(self):
        i = 0
        while True:
            rand = self.random.randrange(1000)
            print(f"{self.name} has counted to {i} and will now sleep for {rand}ms")
            i += 1
            # wait() sleeps like time.sleep, but wakes up early if the event is set.
            if self.stop.wait(rand / 1000):
                print(f"{self.name} interrupted; ending")
                return


def main():
    print("Parent started")
    stop = threading.Event()
    print("Parent is starting Child: Luffy")
    thread_one = threading.Thread(target=Child("Luffy", 1, stop).run)
    thread_one.start()  # start() returns at once; run() goes on in the new thread
    print("Parent is starting Child: Ace")
    thread_two = threading.Thread(target=Child("Ace", 2, stop).run)
    thread_two.start()
    print("Parent will sleep for 3 seconds")
    try:
        threading.Event().wait(3)  # the same as time.sleep(3)
        print("Parent has woken up")
    finally:
        stop.set()  # ask both children to stop
    thread_one.join()  # wait until each child has really finished
    thread_two.join()
    print("Parent ended")
```

Listing 14.2 (Continued) *parent_child.py: the same parent and children in Python; a shared threading.Event replaces Java's interrupt.*

```
if __name__ == "__main__":
    main()
```

Run it with `python3 parent_child.py` (on Windows, `python parent_child.py`):

```
Parent started
Parent is starting Child: Luffy
Parent is starting Child: Ace
Luffy has counted to 0 and will now sleep for 137ms
Ace has counted to 0 and will now sleep for 978ms
Parent will sleep for 3 seconds
Luffy has counted to 1 and will now sleep for 582ms
Luffy has counted to 2 and will now sleep for 867ms
Ace has counted to 1 and will now sleep for 883ms
Luffy has counted to 3 and will now sleep for 821ms
Ace has counted to 2 and will now sleep for 970ms
Luffy has counted to 4 and will now sleep for 782ms
Ace has counted to 3 and will now sleep for 869ms
Parent has woken up
Luffy interrupted; ending
Ace interrupted; ending
Parent ended
```

The structure is identical, and the output shows the same interleaving. The sleep times differ from Java's because Python's random number generator is a different algorithm, but they too are fixed by the seeds.

The one real difference is how the parent asks the children to stop. **Python threads cannot be interrupted.** There is no `interrupt()` method, and a thread inside `time.sleep(5)` will sleep for the full five seconds whatever other threads do. The Python idiom is a `threading.Event`: an object holding one Boolean flag, with a method `set()` that raises the flag and a method `wait(timeout)` that sleeps *until the flag is raised or the timeout passes*, whichever comes first, and returns `True` if the flag was raised. The children call `self.stop.wait(rand / 1000)` instead of `time.sleep`, so they sleep exactly as before but wake up at once when the parent calls `stop.set()`. The parent's own three-second pause, written `threading.Event().wait(3)`, is just a sleep on a fresh event that nobody ever sets.

Notice also `target=Child("Luffy", 1, stop).run`: we pass the *bound method* `run` of a child object, without parentheses. Writing `target=child.run()` would *call* `run` in the parent's thread, which counts forever and brings back the original problem. Section 14.4 returns to this mistake in its Java form.

14.3.2 Stopping with a flag, and the visibility trap

Sometimes the child should stop not because it is interrupted but because it has been *told* to, by a flag it checks on every lap of its loop. The obvious way is a Boolean field that the parent sets to `true` and the child tests. This is also where Java hides one of its least intuitive traps. **Listing 14.3** tests the obvious way against the correct way.

Listing 14.3 `StopFlag.java`: a thread spinning on a plain `boolean` flag may never see the flag change; the same loop on a `volatile` flag stops at once.

```
public class StopFlag {
    static boolean plainStop = false;            // no promise that other threads see
changes
    static volatile boolean volatileStop = false; // every write is visible to every reader

    public static void main(String[] args) throws InterruptedException {
        Thread plain = new Thread(() -> {
            long spins = 0;
            while (!plainStop) {
                spins++;
            }
            System.out.println("plain spinner saw the flag after " + spins + " spins");
        });
        plain.setDaemon(true);  // a daemon thread does not keep the program alive
        plain.start();
        Thread.sleep(500);
        plainStop = true;
        plain.join(1000);       // wait at most one second
        System.out.println("plain flag: still running 1 s later? " + plain.isAlive());

        Thread careful = new Thread(() -> {
            long spins = 0;
            while (!volatileStop) {
                spins++;
            }
            System.out.println("volatile spinner stopped");
        });
        careful.start();
        Thread.sleep(500);
        volatileStop = true;
        careful.join(1000);
        System.out.println("volatile flag: still running 1 s later? " + careful.isAlive());
    }
}
```

```
plain flag: still running 1 s later? true
volatile spinner stopped
volatile flag: still running 1 s later? false
```

The first spinner *never stops*. The main thread set `plainStop = true`, waited a whole second, and the spinner was still running; the program only ends because we made that thread a *daemon* (Section 14.4 explains daemons). The second spinner, identical except that its flag is declared `volatile`, stopped immediately.

How can a thread fail to see a change to a variable? Java's rules do not promise that a write made by one thread is ever seen by another thread unless the two threads *synchronize* in one of a few defined ways. Here the spinning loop reads `plainStop` millions of times and never writes it, so Java's just-in-time compiler is allowed to read the field once, keep the value in a processor register, and turn the loop into `while (true)`. It did exactly that. The keyword **`volatile`** forbids this optimization: every read of a volatile field goes to memory and sees the latest write by any thread [3], [5]. This is the Java *memory model*, and we return to it in Section 14.8.

> **DEFINITION.** A **visibility** problem occurs when one thread changes a shared variable and another thread does not see the change, or sees it late. In Java, a write is guaranteed to be visible to another thread only if the two are connected by a *happens-before* relation: for example, the write is to a `volatile` field and the other thread reads that field, or both threads use the same lock, or one thread starts or joins the other.

Listing 14.4 tries the same experiment in Python.

Listing 14.4 `stop_flag.py`: in CPython a plain attribute used as a stop flag does become visible; a `threading.Event` is still the idiomatic choice.

```
"""stop_flag.py: stopping a busy thread with a plain attribute and with an Event."""
import threading
import time


class Spinner:
    def __init__(self):
        self.stop = False  # a plain attribute, read on every loop

    def run(self):
        spins = 0
        while not self.stop:
            spins += 1
        print("plain spinner saw the flag")


def main():
    s = Spinner()
    t = threading.Thread(target=s.run, daemon=True)
    t.start()
    time.sleep(0.5)
    s.stop = True
    t.join(1)
    print("plain flag: still running 1 s later?", t.is_alive())

    done = threading.Event()  # the idiomatic stop signal

    def careful():
        spins = 0
        while not done.is_set():
            spins += 1
        print("event spinner stopped")

    t2 = threading.Thread(target=careful)
    t2.start()
    time.sleep(0.5)
    done.set()
    t2.join(1)
    print("event: still running 1 s later?", t2.is_alive())


if __name__ == "__main__":
    main()
```

```
plain spinner saw the flag
plain flag: still running 1 s later? False
event spinner stopped
event: still running 1 s later? False
```

Both spinners stopped. In CPython, the standard Python interpreter, every attribute read really does look up the current value, and the Global Interpreter Lock (Section 14.6) makes one thread's writes visible to the others, so the visibility trap of **Listing 14.3** does not appear. That is a fact about today's CPython, not a promise of the language, and a spinning loop wastes a whole core while it waits. The idiomatic Python stop signal is `threading.Event`: `done.set()` to stop, `done.is_set()` to check, and `done.wait(timeout)` to sleep until stopped, as `parent_child.py` did. In Java, use `volatile` for a simple flag, or use interrupts.

14.4 Four Ways to Create a Thread

There is more than one way to hand work to a thread, and each has its place. **Listing 14.5** shows four in Java, using laundry jobs as the work.

Listing 14.5 `CreateThreads.java`: a subclass of `Thread`, a `Runnable`, a lambda, and an `ExecutorService` that returns results through `Future` objects; and what happens if you call `run()` instead of `start()`.

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class CreateThreads {
    // Way 1: extend Thread and override run().
    static class Washer extends Thread {
        Washer(String name) {
            super(name);
        }

        @Override
        public void run() {
            System.out.println(getName() + ": washing (way 1, a subclass of Thread)");
        }
    }

    // Way 2: implement Runnable and hand the object to a Thread.
    static class Dryer implements Runnable {
        @Override
        public void run() {
            String me = Thread.currentThread().getName();
            System.out.println(me + ": drying (way 2, a Runnable)");
        }
    }

    public static void main(String[] args) throws Exception {
        Thread t1 = new Washer("washer-1");
        Thread t2 = new Thread(new Dryer(), "dryer-1");
        // Way 3: a lambda is a Runnable too.
        Thread t3 = new Thread(() -> System.out.println(
                Thread.currentThread().getName() + ": ironing (way 3, a lambda)"), "iron-
1");
        t1.start();
        t1.join();
        t2.start();
        t2.join();
        t3.start();
        t3.join();
```

Listing 14.5 (Continued) *CreateThreads.java: a subclass of Thread, a Runnable, a lambda, and an ExecutorService that returns results through Future objects; and what happens if you call run() instead of start().*

```
        // Way 4: an ExecutorService runs tasks on a pool of reusable threads.
        ExecutorService pool = Executors.newFixedThreadPool(2);
        int[] loadsKg = {5, 7, 3, 6};
        List<Future<Integer>> prices = new ArrayList<>();
        for (int kg : loadsKg) {
            Callable<Integer> job = () -> kg * 7000;  // Rp7,000 per kilogram
            prices.add(pool.submit(job));
        }
        int total = 0;
        for (Future<Integer> price : prices) {
            total += price.get();  // get() waits until that task has finished
        }
        pool.shutdown();           // accept no new tasks; finish the old ones
        System.out.println("pool finished: " + pool.awaitTermination(1, TimeUnit.SECONDS));
        System.out.println("total of 4 loads = Rp" + total);

        // The classic mistake: run() instead of start() creates no new thread.
        Thread notStarted = new Thread(() -> System.out.println(
                "run() executed by: " + Thread.currentThread().getName()), "helper");
        notStarted.run();
        notStarted.start();
        notStarted.join();
    }
}
```

```
washer-1: washing (way 1, a subclass of Thread)
dryer-1: drying (way 2, a Runnable)
iron-1: ironing (way 3, a lambda)
pool finished: true
total of 4 loads = Rp147000
run() executed by: main
run() executed by: helper
```

This output is the same on every run, because after each `start()` the program immediately calls `join()`, so only one extra thread is ever alive at a time. Let us go through the four ways.

Way 1: extend `Thread`. `Washer extends Thread` and overrides `run()`. A `Washer` *is* a thread, so `new Washer("washer-1")` followed by `start()` is enough. The constructor passes a name to `super(name)`, and `getName()` returns it. This is the oldest style and the least flexible: a class that extends `Thread` cannot extend anything else (Chapter 7), and it mixes two ideas, *what* to do and *how* to run it.

Way 2: implement `Runnable`. `Dryer implements Runnable` describes only *what* to do. A separate `Thread` object decides *how* to run it: `new Thread(new Dryer(), "dryer-1")`. Inside `run()`, `Thread.currentThread()` returns the thread that is executing this code, so `Thread.currentThread().getName()` prints `dryer-1`. Separating the work from the thread is the better design, and it is what the Java documentation recommends [6].

Way 3: a lambda. `Runnable` has exactly one abstract method, so it is a *functional interface*, and a lambda expression `() -> …` can stand for it. For a short job this is the most compact form.

Way 4: an `ExecutorService`. Creating a thread costs time and memory, and a program that creates one thread per task soon has thousands. An `ExecutorService` keeps a *pool* of worker threads and feeds tasks to them. `Executors.newFixedThreadPool(2)` makes a pool of two threads. `pool.submit(job)` hands over a task and returns a `Future`, a ticket for a result that may not exist yet. The task here is a `Callable<Integer>`, which is like a `Runnable` except that it returns a value (the price of one load, at Rp7,000 per kilogram). `price.get()` waits until that task has

finished and returns its value. Four loads of 5, 7, 3 and 6 kg cost $21 \times 7000 = 147{,}000$ rupiah in total, as printed. Finally `pool.shutdown()` tells the pool to accept no new tasks and to let its threads end when the queued ones are done, and `awaitTermination` waits for that, returning `true`.

The last block shows the most common beginner's mistake. `notStarted.run()` is an ordinary method call: it executes the lambda *in the current thread*, which is `main`, and no new thread is created. The following `start()` does create one, named `helper`, and the same lambda then reports that it runs there. A program that calls `run()` instead of `start()` usually gives correct answers, just slowly and without any concurrency, which is why the mistake survives so long.

> ⚠ **WARNING.** A thread can be started only once. Calling `start()` a second time on the same `Thread` object throws `IllegalThreadStateException`, even after the first run has finished. To run the same work again, create a new `Thread`, or submit the task to a pool again.

Java 21 and later also offer **virtual threads** (`Thread.ofVirtual().start(task)`), which are managed by the Java runtime, cost very little memory, and make one thread per task affordable even for a million tasks. Races, locks and deadlock work exactly the same way with them; we use ordinary threads so that the listings also run on Java 17.

Listing 14.6 is the Python counterpart.

Listing 14.6 `create_threads.py`: a subclass of `threading.Thread`, a target function, a lambda, and a `ThreadPoolExecutor` with `submit` and `map`.

```
"""create_threads.py: the ways to create a thread in Python."""
import threading
from concurrent.futures import ThreadPoolExecutor


class Washer(threading.Thread):
    """Way 1: subclass Thread and override run()."""

    def __init__(self, name):
        super().__init__(name=name)

    def run(self):
        print(f"{self.name}: washing (way 1, a subclass of Thread)")


def dry():
    """Way 2: any function can be the target of a Thread."""
    print(f"{threading.current_thread().name}: drying (way 2, a target function)")


def main():
    t1 = Washer("washer-1")
    t2 = threading.Thread(target=dry, name="dryer-1")
    # Way 3: a lambda is a function too.
    t3 = threading.Thread(
        target=lambda: print(f"{threading.current_thread().name}: ironing (way 3, a
lambda)"),
        name="iron-1")
    for t in (t1, t2, t3):
        t.start()
        t.join()

    # Way 4: a ThreadPoolExecutor runs tasks on a pool of reusable threads.
    loads_kg = [5, 7, 3, 6]
    with ThreadPoolExecutor(max_workers=2) as pool:
        prices = [pool.submit(lambda kg=kg: kg * 7000) for kg in loads_kg]  # Rp7,000 per
kg
        total = sum(p.result() for p in prices)  # result() waits for that task
        also = sum(pool.map(lambda kg: kg * 7000, loads_kg))
    # Leaving the with block shuts the pool down and waits for its threads.
    print(f"total of 4 loads = Rp{total} (map gives Rp{also})")

    # The classic mistake: run() instead of start() creates no new thread.
    def who():
        print("run() executed by:", threading.current_thread().name)
```

Listing 14.6 (Continued) *create_threads.py: a subclass of threading.Thread, a target function, a lambda, and a ThreadPoolExecutor with submit and map.*

```
    threading.Thread(target=who, name="helper-1").run()  # wrong: runs in main
    right = threading.Thread(target=who, name="helper-2")
    right.start()                                        # right: runs in helper-2
    right.join()


if __name__ == "__main__":
    main()
```

```
washer-1: washing (way 1, a subclass of Thread)
dryer-1: drying (way 2, a target function)
iron-1: ironing (way 3, a lambda)
total of 4 loads = Rp147000 (map gives Rp147000)
run() executed by: MainThread
run() executed by: helper-2
```

The ways correspond one to one. A subclass of `threading.Thread` overrides `run()` and passes its name to `super().__init__(name=name)`; a target function is Python's `Runnable`; a lambda works as a short target. `ThreadPoolExecutor` from `concurrent.futures` [4] is Python's executor, and its `submit` also returns a `Future` whose `result()` waits for the value. `pool.map(fn, items)` is a convenient shortcut that applies `fn` to every item on the pool's threads and yields the results in the original order. Using the executor in a `with` statement shuts it down and waits for its threads when the block ends, the same job as Java's `shutdown` plus `awaitTermination`.

One detail deserves a comment: `lambda kg=kg: kg * 7000`. A lambda defined in a loop captures the *variable* `kg`, not its value at that moment, and by the time the pool runs the lambdas the loop may have moved on, so every lambda would see the last value. Writing `kg=kg` turns the current value into a default argument, which is evaluated immediately. Java avoids the problem by allowing lambdas to capture only variables that are *effectively final*, which is why the Java loop could write `() -> kg * 7000` directly.

The `run()` mistake exists in Python too, with a twist: after `Thread.run()` has executed its target, it deletes the target, so calling `start()` afterwards on the same object would fail. That is why the listing uses two separate thread objects, `helper-1` for the wrong way and `helper-2` for the right way.

Table 14.1 summarizes the vocabulary so far, side by side.

Table 14.1 The basic thread vocabulary of Java and Python.

Task	Java	Python
describe the work	`Runnable` (`run()`), `Callable<T>` (`call()`)	a function (the `target`)
a thread	`new Thread(runnable, name)`	`threading.Thread(target=f, name=n)`
start it	`t.start()`	`t.start()`
wait for it to end	`t.join()`, `t.join(ms)`	`t.join()`, `t.join(seconds)`
is it still running?	`t.isAlive()`, `t.getState()`	`t.is_alive()`
the current thread	`Thread.currentThread()`	`threading.current_thread()`
pause the current thread	`Thread.sleep(ms)`	`time.sleep(seconds)`
ask it to stop	`t.interrupt()`	set a `threading.Event` it checks
background thread	`t.setDaemon(true)` before `start()`	`Thread(..., daemon=True)`
a pool of threads	`ExecutorService`, `Executors.newFixedThreadPool(n)`	`concurrent.futures.ThreadPoolExecutor(n)`
a result that comes later	`Future<T>`, `get()`	`Future`, `result()`

The row about *daemon* threads needs a word. A Java or Python program ends when its last **non-daemon** thread ends. A **daemon** thread is a background helper that does not keep the program alive: when only daemon threads remain, the program exits and they are stopped wherever they are. That makes daemons suitable for background services that may be abandoned (a clock

display, a cache cleaner) and unsuitable for work that must finish (writing a file). We use daemons in this chapter for a different reason: threads that we know are stuck forever, in a deadlock, must not prevent the demonstration program from ending.

14.5 The Life of a Thread

A thread is born, runs, waits in several different ways, and dies. **Figure 14.3** is the classic picture of that life, redrawn after the version found in many Java textbooks [7].

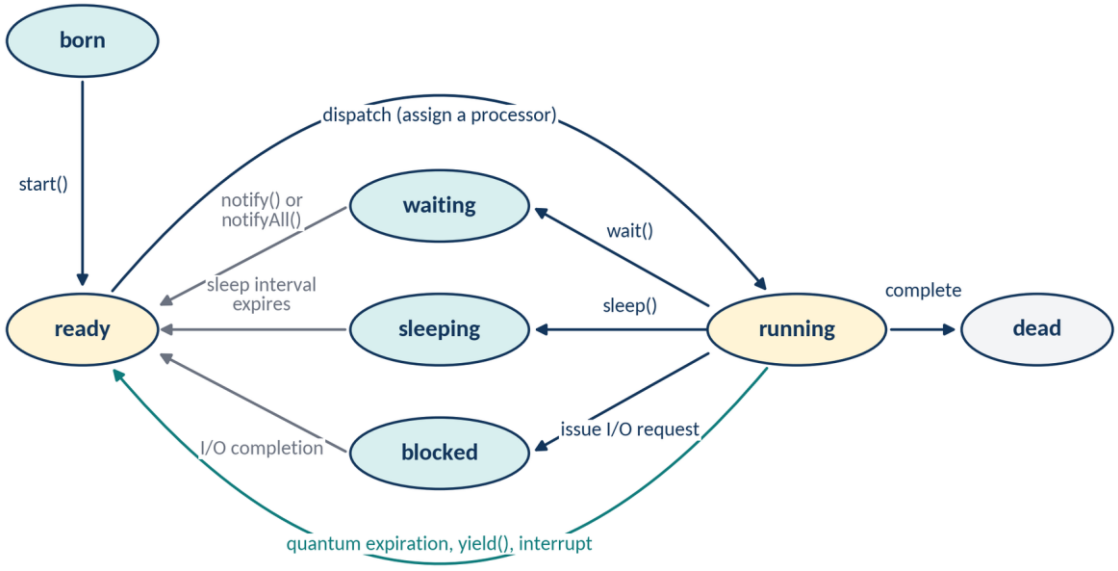


Figure 14.3 The classic life cycle of a Java thread. A thread is born, becomes ready when started, runs when the scheduler dispatches it, and can leave the running state to wait, sleep or block on input and output before it becomes ready again or dies.

The picture separates *ready* (able to run, waiting for a core) from *running* (on a core now). Java itself does not make that distinction visible, because the scheduler switches between the two thousands of times per second. What Java does expose is the enumeration `Thread.State`, with exactly six values, and **Figure 14.4** redraws the life cycle in those terms.

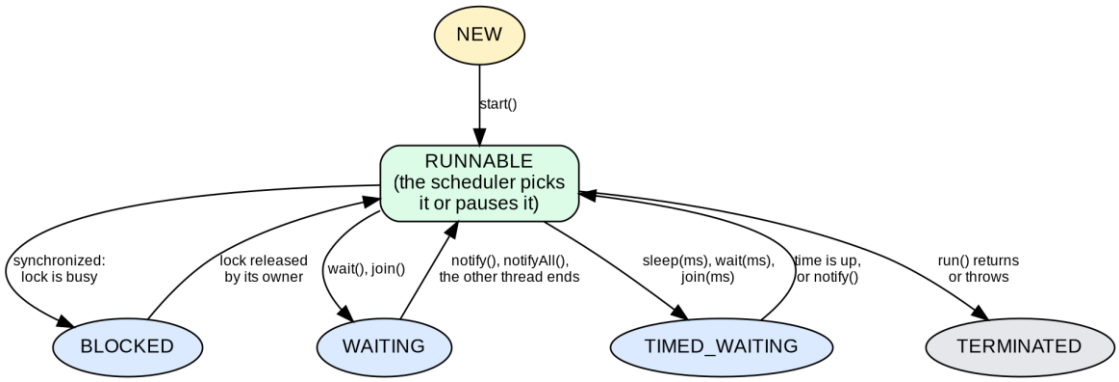


Figure 14.4 The six values of `Thread.State` and the events that move a thread between them. `RUNNABLE` covers both "ready" and "running" of **Figure 14.3**; `BLOCKED` means waiting for a lock, `WAITING` means waiting without a time limit, and `TIMED_WAITING` means waiting with one.

- **`NEW`**: the `Thread` object exists but `start()` has not been called.
- **`RUNNABLE`**: started, and either running or ready to run. A thread that is computing is `RUNNABLE`.
- **`BLOCKED`**: trying to enter a `synchronized` block or method whose lock another thread holds (Section 14.8).

- `WAITING`: waiting, without a time limit, for another thread to do something, for example inside `wait()` (Section 14.9) or `join()`.
- `TIMED_WAITING`: the same, with a time limit: `sleep(ms)`, `wait(ms)`, `join(ms)`.
- `TERMINATED`: `run()` has returned or thrown an exception. A terminated thread cannot be restarted.

Listing 14.7 puts a thread into each state in turn and asks it where it is.

Listing 14.7 `ThreadStates.java`: one thread in each of the six states of `Thread.State`, observed with `getState()`.

```
public class ThreadStates {
    public static void main(String[] args) throws InterruptedException {
        Thread sleeper = new Thread(() -> {
            try {
                Thread.sleep(300);
            } catch (InterruptedException e) {
                return;
            }
        }, "sleeper");
        System.out.println("created, not started : " + sleeper.getState());
        sleeper.start();
        Thread.sleep(100);
        System.out.println("inside sleep(300)    : " + sleeper.getState());
        sleeper.join();
        System.out.println("run() has returned   : " + sleeper.getState());

        Thread busy = new Thread(() -> {
            long n = 0;
            while (!Thread.currentThread().isInterrupted()) {
                n++;  // pure computation: the thread always wants the CPU
            }
        }, "busy");
        busy.start();
        Thread.sleep(100);
        System.out.println("computing            : " + busy.getState());
        busy.interrupt();
        busy.join();

        Object lock = new Object();
        Thread waiter = new Thread(() -> {
            synchronized (lock) {
                try {
                    lock.wait();  // gives the lock back and waits for notify
                } catch (InterruptedException e) {
                    return;
                }
            }
        }, "waiter");
        waiter.start();
        Thread.sleep(100);
        System.out.println("inside wait()        : " + waiter.getState());
```

Listing 14.7 (Continued) *ThreadStates.java: one thread in each of the six states of Thread.State, observed with getState().*

```
        Thread blocked = new Thread(() -> {
            synchronized (lock) {
                System.out.println("blocked thread finally got the lock");
            }
        }, "blocked");
        synchronized (lock) {  // main holds the lock, so "blocked" cannot enter
            blocked.start();
            Thread.sleep(100);
            System.out.println("waiting for a lock   : " + blocked.getState());
            lock.notifyAll();
        }
        waiter.join();
        blocked.join();
        System.out.println("both finished        : " + waiter.getState() + " " +
blocked.getState());
    }
}
```

```
created, not started : NEW
inside sleep(300)    : TIMED_WAITING
run() has returned   : TERMINATED
computing            : RUNNABLE
inside wait()        : WAITING
waiting for a lock   : BLOCKED
blocked thread finally got the lock
both finished        : TERMINATED TERMINATED
```

Each observation is arranged with a short `Thread.sleep(100)` in `main`, which gives the other thread time to reach the state we want to see. (Strictly speaking this is a timing assumption, the kind this chapter warns against; on a very heavily loaded computer a thread might not yet have reached the line we expect after 100 ms. For a demonstration it is reliable, and nothing *breaks* if it is late: the program would merely print a different state.) The `busy` thread computes in a loop and is `RUNNABLE`; its loop condition `!Thread.currentThread().isInterrupted()` lets `busy.interrupt()` stop it even though it never sleeps. The `waiter` calls `lock.wait()` and is `WAITING`. For `BLOCKED`, `main` itself takes the lock with `synchronized (lock)` and then starts a thread that tries to take the same lock; that thread cannot enter and is `BLOCKED` until `main` leaves the block. `lock.notifyAll()` wakes the waiter, and both threads then finish.

Python does not expose a state enumeration. What it offers is `is_alive()` (true between `start()` and the end of `run()`), `ident` (an identifier assigned when the thread starts, `None` before), `threading.enumerate()` (the list of threads that are alive) and, for a lock, `locked()`. **Listing 14.8** observes the same moments with those tools.

Listing 14.8 `thread_states.py`: what Python lets us observe about a thread's life: `is_alive()`, `ident`, `Lock.locked()` and `threading.enumerate()`.

```
"""thread_states.py: what Python lets us observe about a thread's life."""
import threading
import time


def main():
    sleeper = threading.Thread(target=time.sleep, args=(0.3,), name="sleeper")
    print("created, not started : alive =", sleeper.is_alive(), " ident =", sleeper.ident)
    sleeper.start()
    time.sleep(0.1)
    print("inside sleep(0.3)    : alive =", sleeper.is_alive())
    sleeper.join()
    print("run() has returned   : alive =", sleeper.is_alive())

    cond = threading.Condition()

    def wait_for_notify():
        with cond:
            cond.wait()  # gives the lock back and waits for notify

    waiter = threading.Thread(target=wait_for_notify, name="waiter")
    waiter.start()
    time.sleep(0.1)
    print("inside wait()        : alive =", waiter.is_alive())

    lock = threading.Lock()

    def enter():
        with lock:
            print("blocked thread finally got the lock")

    blocked = threading.Thread(target=enter, name="blocked")
    with lock:  # main holds the lock, so "blocked" cannot enter
        blocked.start()
        time.sleep(0.1)
        print("waiting for a lock   : alive =", blocked.is_alive(), " lock held =",
lock.locked())
        with cond:
            cond.notify_all()
    waiter.join()
    blocked.join()
    print("both finished        : alive =", waiter.is_alive(), blocked.is_alive())
    print("threads left:", [t.name for t in threading.enumerate()])


if __name__ == "__main__":
    main()
```

```
created, not started : alive = False  ident = None
inside sleep(0.3)    : alive = True
run() has returned   : alive = False
inside wait()        : alive = True
waiting for a lock   : alive = True  lock held = True
blocked thread finally got the lock
both finished        : alive = False False
threads left: ['MainThread']
```

A Python thread is not alive before `start()`, is alive while it sleeps, waits or is blocked on a lock, and is not alive after `run()` returns. At the end only `MainThread` is left.

14.6 Python's Global Interpreter Lock, and When Threads Help

Before we go looking for trouble, let us see whether threads deliver the speed they promise. **Listing 14.9** counts the prime numbers below three million, first in one thread and then split between two threads, one for each half of the range.

Listing 14.9 `SpeedupDemo.java`: counting primes below three million in one thread and in two threads, with timings.

```
public class SpeedupDemo {
    // Counts the primes in [from, to) by trial division: pure CPU work.
    static int countPrimes(int from, int to) {
        int count = 0;
        for (int n = Math.max(from, 2); n < to; n++) {
            boolean prime = true;
            for (int d = 2; (long) d * d <= n; d++) {
                if (n % d == 0) {
                    prime = false;
                    break;
                }
            }
            if (prime) {
                count++;
            }
        }
        return count;
    }

    public static void main(String[] args) throws InterruptedException {
        final int n = 3_000_000;
        countPrimes(0, n / 10);  // warm-up, so that both timings use compiled code

        long t0 = System.nanoTime();
        int sequential = countPrimes(0, n);
        long t1 = System.nanoTime();

        int[] part = new int[2];
        Thread lower = new Thread(() -> part[0] = countPrimes(0, n / 2));
        Thread upper = new Thread(() -> part[1] = countPrimes(n / 2, n));
        long t2 = System.nanoTime();
        lower.start();
        upper.start();
        lower.join();  // join() also guarantees that we see part[0]
        upper.join();
        long t3 = System.nanoTime();

        System.out.println("processors available : " +
Runtime.getRuntime().availableProcessors());
        System.out.println("primes below " + n + " : " + sequential + " and " + (part[0] +
part[1]));
        System.out.printf("one thread           : %4d ms%n", (t1 - t0) / 1_000_000);
        System.out.printf("two threads          : %4d ms%n", (t3 - t2) / 1_000_000);
    }
}
```

```
processors available : 2
primes below 3000000 : 216816 and 216816
one thread           :  623 ms
two threads          :  602 ms
```

Both versions find 216,816 primes, as they must. The timings, however, are a lesson in honesty. On a computer with two free cores, the two-thread version typically takes a little over half the time of the one-thread version; not exactly half, because the upper half of the range contains larger numbers, which take longer to test, so the two threads do unequal amounts of work. The

run printed above was made on a shared build machine on which the two reported processors were also busy with other jobs, and there the two threads gained almost nothing: 602 ms against 623 ms. Timings of concurrent programs depend on the machine and on what else it is doing, and you should expect different numbers on your own computer. What does not change is the result, 216,816.

How much faster *can* threads make a program? Suppose a fraction f of the running time can be split among p cores and the rest, $1 - f$, cannot. Then the best possible *speedup*, the old time divided by the new time, is given by **Amdahl's law** [8]:

$$S(p) = \frac{1}{(1 - f) + \frac{f}{p}} \tag{14.1}$$

For example, if 90% of the work can be parallelized ($f = 0.9$) and we have $p = 4$ cores, then $S(4) = 1/(0.1 + 0.225) = 1/0.325 \approx 3.08$: four cores give about three times the speed, not four. And however many cores we add, Equation (14.1) can never exceed $1/(1 - f) = 10$. The part that runs one thread at a time dominates in the end. Keep this in mind in Section 14.8: every lock creates a small part that runs one thread at a time.

14.6.1 The Global Interpreter Lock

Now the same experiment in Python. **Listing 14.10** counts primes below 600,000 in one thread, in two threads, and in two *processes*, and then does something quite different: it waits four times for half a second, first in one thread and then in four.

Listing 14.10 `gil_demo.py`: CPU-bound work in one thread, two threads and two processes, and waiting in one thread and four threads.

```
"""gil_demo.py: when Python threads help (waiting) and when they do not (computing)."""
import sys
import threading
import time
from concurrent.futures import ProcessPoolExecutor, ThreadPoolExecutor


def count_primes(bounds):
    """Counts the primes in [lo, hi) by trial division: pure CPU work."""
    lo, hi = bounds
    count = 0
    for n in range(max(lo, 2), hi):
        d = 2
        while d * d <= n:
            if n % d == 0:
                break
            d += 1
        else:
            count += 1
    return count


def wait_for_device(seconds):
    """Stands for waiting on a disk, a network or a washing machine."""
    time.sleep(seconds)
    return seconds


def timed(label, fn):
    t0 = time.perf_counter()
    result = fn()
    print(f"{label:<34}: {time.perf_counter() - t0:5.2f} s  -> {result}")


def main():
    gil = getattr(sys, "_is_gil_enabled", lambda: True)()  # Python 3.13+ can tell
    print(f"Python {sys.version.split()[0]}, GIL enabled: {gil}")
    n = 600_000
    halves = [(0, n // 2), (n // 2, n)]
    timed("CPU work, one thread", lambda: count_primes((0, n)))
    with ThreadPoolExecutor(2) as pool:
        timed("CPU work, two threads", lambda: sum(pool.map(count_primes, halves)))
    with ProcessPoolExecutor(2) as pool:
        timed("CPU work, two processes", lambda: sum(pool.map(count_primes, halves)))
    waits = [0.5, 0.5, 0.5, 0.5]
    timed("four 0.5 s waits, one thread", lambda: sum(map(wait_for_device, waits)))
```

Listing 14.10 (Continued) *gil_demo.py: CPU-bound work in one thread, two threads and two processes, and waiting in one thread and four threads.*

```
    with ThreadPoolExecutor(4) as pool:
        timed("four 0.5 s waits, four threads", lambda: sum(pool.map(wait_for_device,
waits)))


if __name__ == "__main__":
    main()
```

```
Python 3.11.15, GIL enabled: True
CPU work, one thread               :  1.48 s  -> 49098
CPU work, two threads              :  1.25 s  -> 49098
CPU work, two processes            :  1.24 s  -> 49098
four 0.5 s waits, one thread       :  2.00 s  -> 2.0
four 0.5 s waits, four threads     :  0.50 s  -> 2.0
```

The waiting rows are the clearest: four half-second waits take 2.00 s one after another and 0.50 s in four threads. The computing rows need explanation. Splitting the primes between two *threads* did not halve the time in any run we made; on an unloaded multicore computer the two-thread row is typically about the same as, or slightly *slower* than, the one-thread row. The reason has a name.

> **DEFINITION.** The **Global Interpreter Lock (GIL)** is a lock inside the standard Python interpreter, CPython. A thread must hold the GIL to execute Python bytecode, and only one thread can hold it at a time. The interpreter makes the running thread release the GIL regularly (by default every 5 ms) so that the others get a turn, and a thread also releases it whenever it blocks: in `time.sleep`, while reading a file or a socket, while waiting for a lock.

Figure 14.5 shows the consequence. When two threads *compute*, they take turns holding the GIL, so only one of them makes progress at any moment and the total time is the same as, or a little worse than, one thread doing everything. When threads *wait*, a waiting thread holds no GIL, so the others run freely, and many waits overlap perfectly.

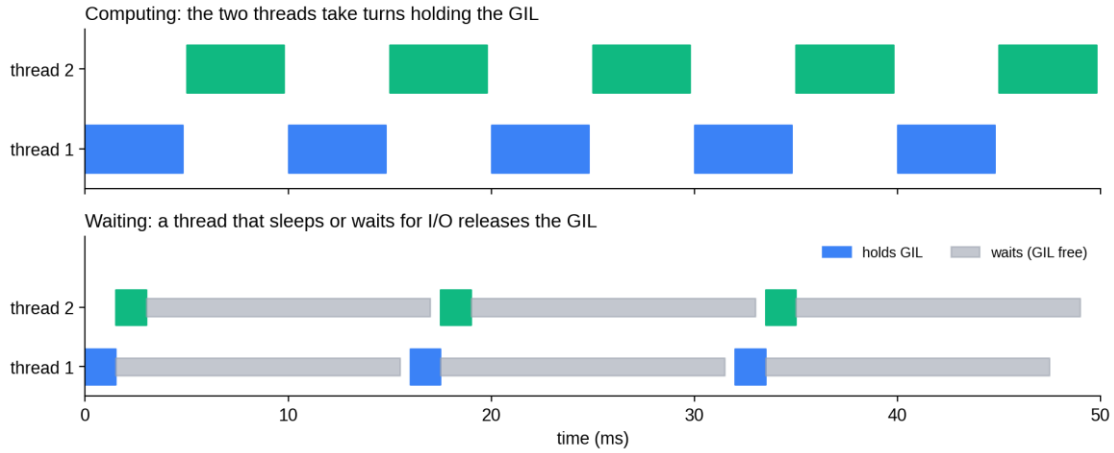


Figure 14.5 The GIL in pictures. Top: two computing threads take turns holding the GIL, so they never compute at the same moment. Bottom: threads that mostly wait release the GIL while waiting, so their waits overlap.

So the rule for CPython is:

- **I/O-bound work** (waiting for disks, networks, databases, sleeping machines): **threads help**, often enormously. Our four waits went from 2.00 s to 0.50 s.
- **CPU-bound work** (arithmetic in pure Python): **threads do not help**. Use `ProcessPoolExecutor` or the `multiprocessing` module instead: each process has its own interpreter and its own GIL, and processes really compute in parallel. On an unloaded two-core computer the process row of **Listing 14.10** is close to half the one-thread row; on the busy machine that produced the output above, even processes gained little. Processes cost more to start than threads and cannot share objects directly (arguments and results are copied between them), which is why `count_primes` receives a small tuple and returns a single number.
- **CPU-bound work inside a library written in C** (NumPy, compression, cryptography): such libraries often release the GIL while they compute, and then threads do help.

Two warnings. First, **the GIL does not make your code thread-safe.** It runs one bytecode instruction at a time, not one *method*; Section 14.7 loses updates in Python in spite of it. Second, the GIL is on its way out. Python 3.13 introduced an experimental *free-threaded* build without it (PEP 703 [9]), and Python 3.14 made that build officially supported, though optional (PEP 779 [10]). There `gil_demo.py` prints `GIL enabled: False` (the function `sys._is_gil_enabled()`

exists from 3.13 on), and the two-thread row speeds up just as Java's does. Code that relies on the GIL by accident may break without it.

14.7 Race Conditions: When Two Threads Share a Counter

Here is a class so simple that it seems impossible to get wrong: a counter that can go up and down.

```
public class Counter {
    private int c = 0;
    public void increment() { c++; }
    public void decrement() { c--; }
    public int get() { return c; }
}
```

Give one counter to three children. Each child increments it and immediately decrements it, a million times. Whatever order the children run in, every increment is followed by its own decrement, so at the end the counter must be 0. And if the children only increment, three children times a million increments must give exactly 3,000,000. **Listing 14.11** tries both experiments, five rounds each, with a fresh counter every round.

Listing 14.11 `RaceDemo.java`: three children share one counter without any lock; five rounds of increment-and-decrement and five rounds of increment only.

```
import java.util.concurrent.CountDownLatch;

public class RaceDemo {
    static class Counter {
        // volatile: every c++ really reads and writes memory (Section 14.8).
        private volatile int c = 0;

        public void increment() {
            c++;
        }

        public void decrement() {
            c--;
        }

        public int get() {
            return c;
        }
    }

    static class Child implements Runnable {
        private final Counter counter;
        private final int times;
        private final boolean alsoDecrement;
        private final CountDownLatch go;

        Child(Counter counter, int times, boolean alsoDecrement, CountDownLatch go) {
            this.counter = counter;
            this.times = times;
            this.alsoDecrement = alsoDecrement;
            this.go = go;
        }

        @Override
        public void run() {
            try {
                go.await();  // all children start at the same moment
            } catch (InterruptedException e) {
                return;
            }
            for (int i = 0; i < times; i++) {
                counter.increment();
                if (alsoDecrement) {
                    counter.decrement();
                }
            }
```

Listing 14.11 (Continued) *RaceDemo.java: three children share one counter without any lock; five rounds of increment-and-decrement and five rounds of increment only.*

```
        }
    }

    static int race(int children, int times, boolean alsoDecrement) throws
InterruptedException {
        Counter counter = new Counter();
        CountDownLatch go = new CountDownLatch(1);
        Thread[] threads = new Thread[children];
        for (int k = 0; k < children; k++) {
            threads[k] = new Thread(new Child(counter, times, alsoDecrement, go));
            threads[k].start();
        }
        go.countDown();  // open the gate
        for (Thread t : threads) {
            t.join();
        }
        return counter.get();
    }

    public static void main(String[] args) throws InterruptedException {
        final int children = 3;
        final int times = 1_000_000;
        System.out.println("Experiment 1: each child increments and decrements " + times +
" times");
        for (int round = 1; round <= 5; round++) {
            System.out.println("  round " + round + ": counter should be 0 and actually is
"
                    + race(children, times, true));
        }
        int expected = children * times;
        System.out.println("Experiment 2: each child increments " + times + " times");
        for (int round = 1; round <= 5; round++) {
            int actual = race(children, times, false);
            System.out.println("  round " + round + ": should be " + expected + ", is " +
actual
                    + ", lost " + (expected - actual));
        }
    }
}
```

One real run printed:

```
Experiment 1: each child increments and decrements 1000000 times
  round 1: counter should be 0 and actually is 0
  round 2: counter should be 0 and actually is 0
  round 3: counter should be 0 and actually is 0
  round 4: counter should be 0 and actually is 399
  round 5: counter should be 0 and actually is 249
Experiment 2: each child increments 1000000 times
  round 1: should be 3000000, is 2423058, lost 576942
  round 2: should be 3000000, is 2924583, lost 75417
  round 3: should be 3000000, is 1020932, lost 1979068
  round 4: should be 3000000, is 3000000, lost 0
  round 5: should be 3000000, is 3000000, lost 0
```

The counter that “must be 0” ended at 399 and 249 in two of the five rounds. The counter that “must be 3,000,000” was short by 576,942 in round 1 and by almost two million in round 3: two thirds of all the increments of that round vanished. No exception was thrown, no warning was printed, and in rounds 4 and 5 the answer was perfect. This is a **race condition**, and the fact that it sometimes gives the right answer is exactly what makes it so dangerous.

> **DEFINITION.** A **race condition** is a bug in which the result of a program depends on the relative timing of its threads, so that some interleavings give a wrong result. The most common kind is the **lost update**: two threads read the same old value of a shared variable, each computes a new value from it, and the second write overwrites the first.

What varies here, and what does not? The number of lost updates varies wildly between rounds and between runs, and some rounds lose nothing. We ran the program ten times: in seven of the ten runs at least one of the increment-only rounds lost updates, and in eight of the ten at least one increment-and-decrement round ended away from 0; in the other three runs all five increment-only rounds were exact, and one run was perfect in both experiments. On a computer with more free cores the losses are usually larger and more frequent. What never varies is the *direction*: a lost update can only make the increment-only counter too small, never too large, and the correct total is only guaranteed when no two updates overlap.

14.7.1 Why `c++` is not one step

The trouble is that `c++` is not one operation. Ask the JDK's disassembler, `javap -c`, what the compiler produced for `increment()` in the nested class `Counter` (the chapter's README gives the exact command):

```
public void increment();
  Code:
     0: aload_0
     1: dup
     2: getfield      #7                  // Field c:I
     5: iconst_1
     6: iadd
     7: putfield      #7                  // Field c:I
    10: return
```

Four instructions do the work. `getfield` **reads** the current value of `c` from the object into the thread's own working area (its *operand stack*); `iconst_1` and `iadd` **add** one to that private copy; `putfield` **writes** the result back into the object. Read, modify, write: three steps, and the scheduler may pause a thread between any two of them, while other threads run and change `c`.

Figure 14.6 slows the collision down to single steps, with two children, one doing `c++` then `c--`, the other the same.

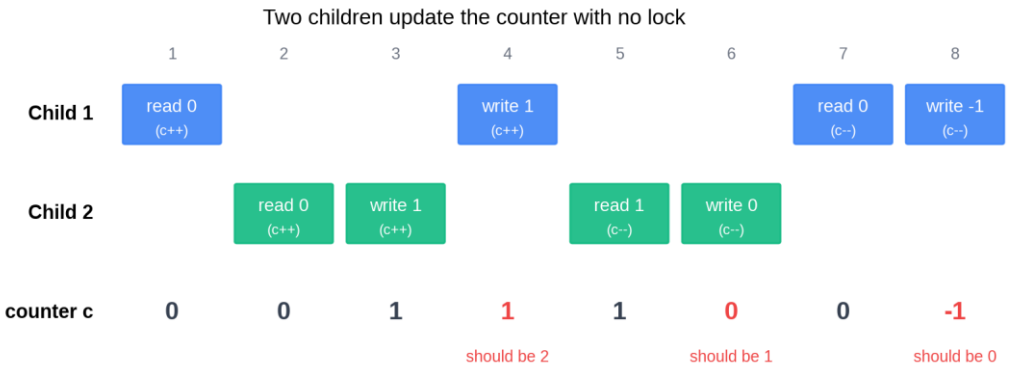


Figure 14.6 A lost update, step by step. Child 1 reads 0; Child 2 reads 0 and writes 1; Child 1 writes its own 1 over it. Two increments have happened and the counter says 1. Two more steps later a decrement is lost as well, and the counter ends at −1 instead of 0.

In words, the correct flow of execution would be: Child 1 increments, Child 1 decrements, Child 2 increments, and so on, each operation complete before the next begins. What actually can happen is:

1. Child 1 reads the counter: 0.
2. Child 2 reads the counter: 0.
3. Child 2 writes 1 (correct so far: one increment done).
4. Child 1 also writes 1, computed from the 0 it read in step 1. It should be 2: Child 2's increment is lost.
5. Child 2 reads 1.
6. Child 2 writes 0. It should be 1.
7. Child 1 reads 0.
8. Child 1 writes -1. It should be 0.

Each thread did exactly what its code said; the error lives entirely in the interleaving.

How likely is a bad interleaving? Count them. If one thread executes m steps and another executes n steps, then every possible execution is an ordering of all $m + n$ steps that keeps each thread's own steps in order, and there are

$$\binom{m+n}{m} = \frac{(m+n)!}{m!\,n!} \tag{14.2}$$

such orderings. For two increments of three steps each ($m = n = 3$), Equation (14.2) gives $\binom{6}{3} = 20$ interleavings. Only 2 of them are correct: all of thread A's steps before all of thread B's, or the reverse. The other 18 lose an update. In a real program each thread executes millions of steps, the number of interleavings is astronomically large, and the scheduler picks one of them for each run. Testing can show that a race exists; it can never show that one does *not*, because the next run may pick an interleaving that no test has ever tried.

For the increment-only experiment we can say precisely what is lost. With T threads each performing N increments, the number of lost updates in a run is

$$L = T \cdot N - c_{\text{end}} \tag{14.3}$$

where c_{end} is the value of the counter when all threads have finished; **Listing 14.11** prints L as `lost`. For round 1 above, Equation (14.3) gives $L = 3 \cdot 1{,}000{,}000 - 2{,}423{,}058 = 576{,}942$.

One more remark about **Listing 14.11**: its counter field is declared `volatile`. Section 14.3.2 showed that without `volatile` Java may keep a field in a register; in this program that would let the compiler turn a million `c++` into one `c += 1000000` done at the end of the loop, which loses updates in huge blocks or, when the threads happen not to overlap, not at all. `volatile` makes every `c++` a genuine read and a genuine write of memory, so the demonstration shows the race in its textbook form. And it proves a point that surprises many programmers: **`volatile` does not make `c++` safe.** It guarantees that each read sees the latest write; it does not stop another thread from writing between our read and our write.

14.7.2 The same race in Python

Listing 14.12 repeats the increment-only experiment in Python, first with a plain `self.c += 1`, then with the same three steps written out and a pause between the read and the write.

Listing 14.12 `race_demo.py`: lost updates on a shared counter in Python, with a plain `+=` and with a deliberately yielding read-modify-write.

```
"""race_demo.py: lost updates on a shared counter."""
import threading
import time


class Counter:
    def __init__(self):
        self.c = 0

    def increment(self):
        self.c += 1  # read, add one, write back: three steps

    def get(self):
        return self.c


class SlowCounter(Counter):
    """The same three steps, with a pause between the read and the write."""

    def increment(self):
        value = self.c        # 1. read
        time.sleep(0)         # let another thread run right here
        self.c = value + 1    # 2.-3. add one and write back


def race(counter, children, times):
    def child():
        for _ in range(times):
            counter.increment()

    threads = [threading.Thread(target=child) for _ in range(children)]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    return counter.get()
```

Listing 14.12 (Continued) *race_demo.py: lost updates on a shared counter in Python, with a plain += and with a deliberately yielding read-modify-write.*

```
def main():
    children, times = 3, 200_000
    expected = children * times
    print(f"Experiment 1: {children} children, {times} plain increments each")
    for rnd in range(1, 4):
        actual = race(Counter(), children, times)
        print(f"  round {rnd}: should be {expected}, is {actual}, lost {expected -
actual}")
    times = 2_000
    expected = children * times
    print(f"Experiment 2: {children} children, {times} increments with a pause each")
    for rnd in range(1, 4):
        actual = race(SlowCounter(), children, times)
        print(f"  round {rnd}: should be {expected}, is {actual}, lost {expected -
actual}")


if __name__ == "__main__":
    main()
```

```
Experiment 1: 3 children, 200000 plain increments each
  round 1: should be 600000, is 600000, lost 0
  round 2: should be 600000, is 600000, lost 0
  round 3: should be 600000, is 600000, lost 0
Experiment 2: 3 children, 2000 increments with a pause each
  round 1: should be 6000, is 2001, lost 3999
  round 2: should be 6000, is 2000, lost 4000
  round 3: should be 6000, is 2008, lost 3992
```

The honest result first: in Experiment 1, with a plain `self.c += 1`, CPython 3.11 lost **nothing**, in these three rounds and in every other run we made. Does that mean `+=` is atomic in Python? Look at the bytecode, printed with `dis.dis(Counter.increment)`:

```
 10           0 RESUME                   0

 11           2 LOAD_FAST                0 (self)
              4 COPY                     1
              6 LOAD_ATTR                0 (c)
             16 LOAD_CONST               1 (1)
             18 BINARY_OP               13 (+=)
             22 SWAP                     2
             24 STORE_ATTR               0 (c)
             34 LOAD_CONST               0 (None)
             36 RETURN_VALUE
```

The shape is the same as Java's: `LOAD_ATTR` reads `c`, `BINARY_OP (+=)` adds one, `STORE_ATTR` writes the result back. The language does not promise that these steps run together. They happen to, in CPython 3.10 and later, because the interpreter only considers handing the GIL to another thread at certain points (backward jumps and calls, roughly), and none of them falls inside this short sequence. Older versions of CPython, other Python implementations, and the free-threaded build of Section 14.6.1 can and do lose such updates [9]. The bytecode shown also differs a little between versions (Python 3.12 and 3.13 print different instruction names), but the read, add, write shape is always there.

Experiment 2 makes the hazard visible on demand. `SlowCounter.increment` performs the same three steps, but calls `time.sleep(0)` between the read and the write. `sleep(0)` sleeps for no time at all; its only effect is to release the GIL and let another thread run, which is exactly what the scheduler is always *allowed* to do at that point. It does not create a new bug; it merely opens the window of the old one wide enough to see. The result is dramatic: of 6,000 increments, about 4,000 were lost in every round. With three threads taking turns at the pause, almost every read is followed by two other threads reading the same value before anyone writes, so two out of every three increments vanish.

> ⚠ **WARNING.** "It worked when I tested it" proves nothing about concurrent code. A race that loses one update in a million operations will pass every test you write in class and then corrupt a real ledger once a day in production. The only defense is to find every piece of data that more than one thread can change and to protect it on purpose, as the next section shows.

14.8 Mutual Exclusion: `synchronized`, Locks, and Atomics

The cure for the lost update is to make the three steps *indivisible* as far as other threads are concerned: while one thread is between its read and its write, no other thread may touch the counter. The stretch of code that must run without interference is called a **critical section**, and the rule that at most one thread may be inside a given critical section at a time is called **mutual exclusion**.

> **DEFINITION.** A **lock** (also *mutex*, from *mutual exclusion*) is an object that at most one thread can hold at a time. A thread **acquires** the lock before entering a critical section and **releases** it after leaving. A thread that tries to acquire a lock held by another thread waits (in Java its state is `BLOCKED` or `WAITING`) until the lock is released.

In Java, **every object has a built-in lock**, called its *intrinsic lock* or *monitor* [11]. The keyword `synchronized` uses it in two forms.

A **synchronized method** acquires the lock of `this` object when it is called and releases it when it returns, even if it returns by throwing an exception:

```
public synchronized void increment() {
    c++;
}
```

A **synchronized block** acquires the lock of any object you name, for the statements inside the braces:

```
synchronized (counter) {
    counter.increment();
}
```

The first form lets the counter protect itself; the second works only if *all* callers remember it. Either way, the rule is: **every access to shared data that some thread changes must hold the same lock**, readers included; that is why `get()` below is synchronized too.

Besides `synchronized`, the package `java.util.concurrent` offers explicit lock objects and atomic variables [6]. **Listing 14.13** runs the increment-only experiment of **Listing 14.11** with five different cures.

Listing 14.13 `SafeCounters.java`: five correct counters: synchronized methods, a synchronized block in the caller, a `ReentrantLock`, an `AtomicInteger` and a `LongAdder`.

```
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.LongAdder;
import java.util.concurrent.locks.ReentrantLock;

public class SafeCounters {
    interface Counter {
        void increment();

        long get();
    }

    // Cure 1: synchronized methods lock the counter object itself.
    static class SyncMethodCounter implements Counter {
        private int c = 0;

        public synchronized void increment() {
            c++;
        }

        public synchronized long get() {
            return c;
        }
    }

    // Cure 2: the caller locks the object with a synchronized block (see race()).
    static class PlainCounter implements Counter {
        private int c = 0;

        public void increment() {
            c++;
        }

        public long get() {
            return c;
        }
    }

    // Cure 3: an explicit lock, released in finally.
    static class LockCounter implements Counter {
        private final ReentrantLock lock = new ReentrantLock();
        private int c = 0;
```

Listing 14.13 (Continued) *SafeCounters.java: five correct counters: synchronized methods, a synchronized block in the caller, a ReentrantLock, an AtomicInteger and a LongAdder.*

```
        public void increment() {
            lock.lock();
            try {
                c++;
            } finally {
                lock.unlock();
            }
        }

        public long get() {
            lock.lock();
            try {
                return c;
            } finally {
                lock.unlock();
            }
        }
    }

    // Cure 4: an atomic variable; the hardware does read-add-write in one step.
    static class AtomicCounter implements Counter {
        private final AtomicInteger c = new AtomicInteger();

        public void increment() {
            c.incrementAndGet();
        }

        public long get() {
            return c.get();
        }
    }

    // Cure 5: a LongAdder spreads the count over cells; best when many threads add.
    static class AdderCounter implements Counter {
        private final LongAdder c = new LongAdder();

        public void increment() {
            c.increment();
        }

        public long get() {
            return c.sum();
        }
    }
```

Listing 14.13 (Continued) *SafeCounters.java: five correct counters: synchronized methods, a synchronized block in the caller, a ReentrantLock, an AtomicInteger and a LongAdder.*

```
    static void race(String label, Counter counter, boolean callerLocks)
            throws InterruptedException {
        final int children = 3;
        final int times = 1_000_000;
        Runnable child = () -> {
            for (int i = 0; i < times; i++) {
                if (callerLocks) {
                    synchronized (counter) {
                        counter.increment();
                    }
                } else {
                    counter.increment();
                }
            }
        };
        Thread[] threads = new Thread[children];
        long t0 = System.nanoTime();
        for (int k = 0; k < children; k++) {
            threads[k] = new Thread(child);
            threads[k].start();
        }
        for (Thread t : threads) {
            t.join();
        }
        long ms = (System.nanoTime() - t0) / 1_000_000;
        System.out.printf("%-22s: should be %d, is %d  (%d ms)%n",
                label, children * times, counter.get(), ms);
    }

    public static void main(String[] args) throws InterruptedException {
        race("synchronized methods", new SyncMethodCounter(), false);
        race("synchronized block", new PlainCounter(), true);
        race("ReentrantLock", new LockCounter(), false);
        race("AtomicInteger", new AtomicCounter(), false);
        race("LongAdder", new AdderCounter(), false);
    }
}
```

```
synchronized methods  : should be 3000000, is 3000000  (52 ms)
synchronized block    : should be 3000000, is 3000000  (69 ms)
ReentrantLock         : should be 3000000, is 3000000  (140 ms)
AtomicInteger         : should be 3000000, is 3000000  (81 ms)
LongAdder             : should be 3000000, is 3000000  (76 ms)
```

Every counter now reaches exactly 3,000,000, on every run. The timings in parentheses vary a great deal between runs and machines and should not be read as a benchmark; the Java runtime optimizes uncontended locks very aggressively, and on this busy two-core machine the ranking changed from run to run. The correct totals do not vary. Here is what each cure does.

- **`synchronized` methods** (`SyncMethodCounter`) are the simplest cure and the right default for a small class whose methods are short.
- **A `synchronized` block in the caller** (`PlainCounter` together with `synchronized (counter)` in `race`) protects a class that was not written to be thread-safe. It is correct here only because *every* call goes through the block.
- **`ReentrantLock`** (`LockCounter`) is an explicit lock object with `lock()` and `unlock()`. The pattern is always the same: `lock()`, then `try`, then the critical section, then `finally { unlock(); }`, so that an exception cannot leave the lock held forever. An explicit lock can do things `synchronized` cannot: `tryLock()` gives up if the lock is busy, `tryLock(time, unit)` gives up after a timeout

(we will need that against deadlock in Section 14.12), and `new ReentrantLock(true)` makes a *fair* lock that serves waiting threads in arrival order (Section 14.13).

- **`AtomicInteger`** (`AtomicCounter`) needs no lock at all. `incrementAndGet()` performs read, add and write as **one indivisible hardware instruction** (a *compare-and-swap* loop): the processor itself guarantees that no other core can change the value between the read and the write. Atomic classes exist for `int`, `long`, Booleans and references.
- **`LongAdder`** (`AdderCounter`) keeps several internal cells and lets different threads add to different cells, summing them only when `sum()` is called. When many threads increment very often, this is faster than one `AtomicLong`, because the threads do not fight over a single memory location.

> **DEFINITION.** An operation is **atomic** if no other thread can ever observe it half done: it has either not happened at all or happened completely. `c++` on an `int` is not atomic; `incrementAndGet()` on an `AtomicInteger` is; and any sequence of statements becomes atomic with respect to other threads that use the same lock.

14.8.1 Why a lock also fixes visibility

A lock does two jobs, not one. Besides keeping other threads out, releasing a lock *publishes* every write the thread made while holding it, and acquiring the same lock later *sees* all of them. In the vocabulary of Section 14.3.2, the release of a lock happens-before every later acquisition of the same lock [3], [5]. That is why `synchronized long get()` is necessary: without it, a thread calling `get()` might see a stale value of `c`, exactly as the spinner of **Listing 14.3** saw a stale flag. The same guarantee explains why `SpeedupDemo` could read `part[0]` after `lower.join()`: everything a thread did happens-before another thread's successful `join()` on it.

Java locks are also **reentrant**: a thread that already holds a lock may acquire it again without blocking itself, and must release it as many times as it acquired it. So a synchronized method may call another synchronized method on the same object. Python distinguishes the two kinds explicitly: `threading.Lock` is *not* reentrant (a thread that acquires it twice waits for itself forever), and `threading.RLock` is.

14.8.2 Locks in Python

Listing 14.14 protects the slow counter of **Listing 14.12** with a `threading.Lock`, keeping the provocative `sleep(0)` inside the critical section.

Listing 14.14 `safe_counters.py`: the slow read-modify-write counter made correct with `threading.Lock`, written with `with` and with explicit `acquire`/`release`.

```
"""safe_counters.py: the lost updates of race_demo.py, cured with a Lock."""
import threading
import time


class LockedCounter:
    def __init__(self):
        self._c = 0
        self._lock = threading.Lock()

    def increment(self):
        with self._lock:          # only one thread at a time past this line
            value = self._c
            time.sleep(0)         # the same pause as SlowCounter: now harmless
            self._c = value + 1

    def get(self):
        with self._lock:
            return self._c


class ExplicitLockCounter(LockedCounter):
    """The with statement above is shorthand for acquire ... try ... finally release."""

    def increment(self):
        self._lock.acquire()
        try:
            value = self._c
            time.sleep(0)
            self._c = value + 1
        finally:
            self._lock.release()  # released even if the body raises


def race(counter, children, times):
    def child():
        for _ in range(times):
            counter.increment()
```

Listing 14.14 (Continued) *safe_counters.py: the slow read-modify-write counter made correct with threading.Lock, written with with and with explicit acquire/release.*

```
    threads = [threading.Thread(target=child) for _ in range(children)]
    t0 = time.perf_counter()
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    return counter.get(), time.perf_counter() - t0


def main():
    children, times = 3, 2_000
    for cls in (LockedCounter, ExplicitLockCounter):
        value, secs = race(cls(), children, times)
        print(f"{cls.__name__:<20}: should be {children * times}, is {value}"
              f"  ({secs:.2f} s)")


if __name__ == "__main__":
    main()
```

```
LockedCounter       : should be 6000, is 6000  (0.42 s)
ExplicitLockCounter : should be 6000, is 6000  (0.42 s)
```

Both counters reach exactly 6,000 on every run. The `sleep(0)` still hands the GIL to another thread in the middle of the update, but that thread then tries to acquire `self._lock`, finds it held, and waits; the update is completed before anyone else can read `_c`. The `with self._lock:` statement acquires the lock on entry and releases it on exit, even if the body raises an exception. `ExplicitLockCounter` spells out what `with` does: `acquire()`, then `try` … `finally: release()`, exactly the Java `ReentrantLock` pattern. Prefer `with`; it cannot forget the release.

Python's standard library has no general atomic integer class. For a counter, use a lock as above, or give each thread its own local count and add them together after `join()`, which needs no sharing at all. **Table 14.2** collects the synchronization tools of this chapter.

Table 14.2 Synchronization tools of Java and Python used in this chapter.

Need	Java	Python
mutual exclusion	`synchronized`, `ReentrantLock`	`threading.Lock`, `threading.RLock`
lock, but give up if busy	`tryLock()`, `tryLock(t, unit)`	`acquire(blocking=False)`, `acquire(timeout=t)`
single variable, no lock	`AtomicInteger`, `AtomicLong`, `LongAdder`	(none; use a lock)
always see the latest value	`volatile` field	(CPython attributes behave so; do not rely on it)
wait until a condition holds	`wait`/`notifyAll`, `Condition`	`threading.Condition`
a one-shot signal	`CountDownLatch`	`threading.Event`
at most k threads at once	`Semaphore(k)`	`threading.Semaphore(k)`
hand data between threads	`BlockingQueue`	`queue.Queue`

✓ **KEY INSIGHT.** The best lock is the one you do not need. Data that is never shared (local variables, per-thread results combined after `join()`) cannot race, and data that never changes (the immutable objects of Chapter 13) can be shared by any number of threads without any lock at all. Lock what must be shared *and* changed, and only that.

14.9 Waiting for a Condition: `wait`, `notifyAll`, and `Condition`

Locks keep threads apart. Often, however, threads must also *wait for each other*: a washing machine with nothing to do must wait until an order arrives; a front desk with a full basket must wait until a machine takes an order out. This is the classic **producer–consumer** problem: one thread produces values, another consumes them, and between them is a **buffer** of limited size. The producer must wait while the buffer is full, and the consumer must wait while it is empty.

Our buffer has five cells and is used *circularly*: after writing into cell 4, the producer continues at cell 0 again, provided the consumer has emptied it. Two indexes, `writeLocation` and `readLocation`, say where the next value goes and where the next value comes from, and each moves forward by one, wrapping around, as

$$w \leftarrow (w + 1) \bmod n, \qquad r \leftarrow (r + 1) \bmod n \tag{14.4}$$

with $n = 5$ cells. When the two indexes become equal, the buffer is either completely full (if the producer just moved) or completely empty (if the consumer just moved); two Boolean flags,

`writeable` and `readable`, remember which. **Figure 14.7** shows the full state that we are about to meet in a real run.

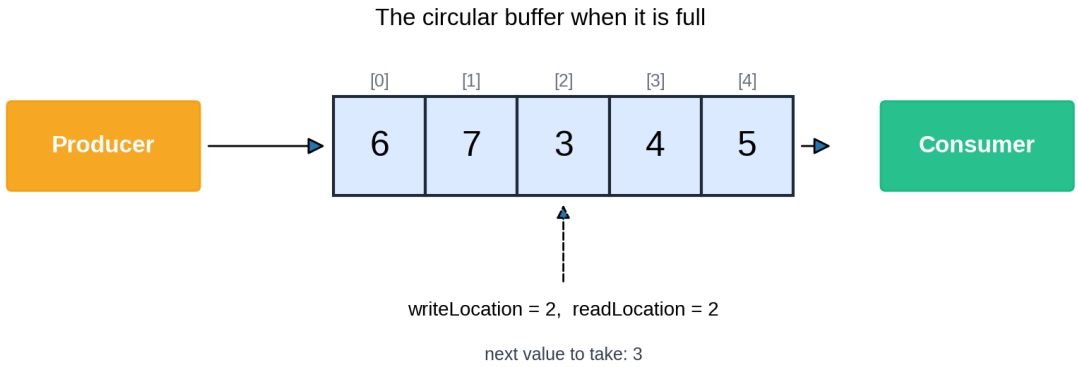


Figure 14.7 The five-cell circular buffer when it is full. Both indexes point to cell 2: the consumer's next value is the 3 stored there, and the producer's next value will go there as soon as the 3 has been taken.

How should a thread wait? A loop such as `while (!writeable) { }` would work in a sense, but it burns a whole core doing nothing, and, worse, it would spin while *holding the lock*, so the consumer could never get in to make room. What we need is a way to say: "release the lock, sleep until somebody tells me that things have changed, then take the lock back." Every Java object provides exactly that, together with its intrinsic lock [11]:

- **`wait()`**, called while holding an object's lock, releases that lock and puts the thread to sleep (state `WAITING`) in the object's *wait set*. When the thread is woken, it first re-acquires the lock, and only then does `wait()` return.
- **`notify()`** wakes one thread from the object's wait set, if there is one; **`notifyAll()`** wakes all of them. The caller must hold the lock, and the woken threads cannot actually continue until the caller releases it.

This combination of a lock with a waiting room is called a **monitor**, an idea due to Hoare [12]. **Listing 14.15** uses it for our circular buffer, adapted from a classic textbook example [7].

Listing 14.15 `ProCon.java`: a producer and a slower consumer share a five-cell circular buffer; `wait()` and `notifyAll()` make each wait for the other.

```
import java.util.Arrays;
import java.util.Random;

public class ProCon {
    // A circular buffer of five cells shared by one producer and one consumer.
    static class HoldIntegerSynchronized {
        private final int[] sharedInt = {-1, -1, -1, -1, -1};
        private boolean writeable = true;   // is there a free cell?
        private boolean readable = false;   // is there a full cell?
        private int readLocation = 0;
        private int writeLocation = 0;

        public synchronized void setSharedInt(int value) throws InterruptedException {
            while (!writeable) {            // while, not if: re-check after every wake-up
                System.out.println("WAITING TO PRODUCE " + value);
                wait();                     // releases the lock while waiting
            }
            sharedInt[writeLocation] = value;
            readable = true;
            System.out.printf("Produced %2d into cell %d", value, writeLocation);
            writeLocation = (writeLocation + 1) % sharedInt.length;
            System.out.printf("   write %d  read %d  buffer: %s%n",
                    writeLocation, readLocation, Arrays.toString(sharedInt));
            if (writeLocation == readLocation) {
                writeable = false;
                System.out.println("BUFFER FULL");
            }
            notifyAll();                    // wake the consumer if it is waiting
        }

        public synchronized int getSharedInt() throws InterruptedException {
            while (!readable) {
                System.out.println("WAITING TO CONSUME");
                wait();
            }
            writeable = true;
            int value = sharedInt[readLocation];
            System.out.printf("Consumed %2d from cell %d", value, readLocation);
            readLocation = (readLocation + 1) % sharedInt.length;
            System.out.printf("   write %d  read %d  buffer: %s%n",
                    writeLocation, readLocation, Arrays.toString(sharedInt));
            if (readLocation == writeLocation) {
                readable = false;
                System.out.println("BUFFER EMPTY");
            }
            notifyAll();                    // wake the producer if it is waiting
```

Listing 14.15 (Continued) *ProCon.java: a producer and a slower consumer share a five-cell circular buffer; wait() and notifyAll() make each wait for the other.*

```
            return value;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        HoldIntegerSynchronized buffer = new HoldIntegerSynchronized();
        Thread producer = new Thread(() -> {
            Random r = new Random(1);
            try {
                for (int i = 1; i <= 10; i++) {
                    Thread.sleep(r.nextInt(200));  // time to make the next value
                    buffer.setSharedInt(i);
                }
            } catch (InterruptedException e) {
                return;
            }
            System.out.println("Producer finished producing values");
        });
        Thread consumer = new Thread(() -> {
            Random r = new Random(7);
            int sum = 0;
            try {
                for (int i = 1; i <= 10; i++) {
                    Thread.sleep(r.nextInt(700));  // the consumer is slower
                    sum += buffer.getSharedInt();
                }
            } catch (InterruptedException e) {
                return;
            }
            System.out.println("Consumer retrieved values totaling: " + sum);
        });
        producer.start();
        consumer.start();
        producer.join();
        consumer.join();
    }
}
```

```
WAITING TO CONSUME
Produced  1 into cell 0   write 1  read 0  buffer: [1, -1, -1, -1, -1]
Consumed  1 from cell 0   write 1  read 1  buffer: [1, -1, -1, -1, -1]
BUFFER EMPTY
Produced  2 into cell 1   write 2  read 1  buffer: [1, 2, -1, -1, -1]
Produced  3 into cell 2   write 3  read 1  buffer: [1, 2, 3, -1, -1]
Consumed  2 from cell 1   write 3  read 2  buffer: [1, 2, 3, -1, -1]
Produced  4 into cell 3   write 4  read 2  buffer: [1, 2, 3, 4, -1]
Produced  5 into cell 4   write 0  read 2  buffer: [1, 2, 3, 4, 5]
Produced  6 into cell 0   write 1  read 2  buffer: [6, 2, 3, 4, 5]
Produced  7 into cell 1   write 2  read 2  buffer: [6, 7, 3, 4, 5]
BUFFER FULL
WAITING TO PRODUCE 8
Consumed  3 from cell 2   write 2  read 3  buffer: [6, 7, 3, 4, 5]
Produced  8 into cell 2   write 3  read 3  buffer: [6, 7, 8, 4, 5]
BUFFER FULL
Consumed  4 from cell 3   write 3  read 4  buffer: [6, 7, 8, 4, 5]
Produced  9 into cell 3   write 4  read 4  buffer: [6, 7, 8, 9, 5]
BUFFER FULL
Consumed  5 from cell 4   write 4  read 0  buffer: [6, 7, 8, 9, 5]
Produced 10 into cell 4   write 0  read 0  buffer: [6, 7, 8, 9, 10]
BUFFER FULL
Producer finished producing values
Consumed  6 from cell 0   write 0  read 1  buffer: [6, 7, 8, 9, 10]
Consumed  7 from cell 1   write 0  read 2  buffer: [6, 7, 8, 9, 10]
Consumed  8 from cell 2   write 0  read 3  buffer: [6, 7, 8, 9, 10]
Consumed  9 from cell 3   write 0  read 4  buffer: [6, 7, 8, 9, 10]
Consumed 10 from cell 4   write 0  read 0  buffer: [6, 7, 8, 9, 10]
BUFFER EMPTY
Consumer retrieved values totaling: 55
```

Follow the story told by the output. The consumer happened to be ready first and found the buffer empty, so it printed `WAITING TO CONSUME` and waited. The producer's first value woke it; it took 1, and the buffer was empty again. Then the producer, which is faster, got ahead: it wrote 2 to 7 while the consumer took only the 2, wrapped around from cell 4 to cells 0 and 1 as Equation (14.4) prescribes, and at 7 the write index caught up with the read index: `BUFFER FULL`. Trying to write 8, the producer found no free cell, printed `WAITING TO PRODUCE 8` and waited. Each time the consumer took a value, its `notifyAll()` woke the producer, which filled the freed cell at once, so the buffer stayed full until the producer finished at 10. The consumer then drained the remaining five values, the buffer became empty, and the consumer's total is $1 + 2 + \cdots + 10 = 55$.

The array keeps showing consumed values, such as the 3 in `[6, 7, 3, 4, 5]`: consuming only moves `readLocation`, and the old number stays until overwritten. That is why the flags, not the contents, say whether the buffer is full or empty.

What varies from run to run? The seeds 1 and 7 fix the sleep times, so most runs print the same lines; but when a producer's and a consumer's wake-ups fall within a millisecond of each other, two lines can swap and a `WAITING` line can appear or disappear. What never varies: the values are consumed in the order 1, 2, …, 10, each exactly once, the buffer never holds more than five, and the total is 55.

Now the details of `setSharedInt` and `getSharedInt`, which are the heart of the matter.

- Both methods are `synchronized`, so they hold the buffer's lock while they inspect and change its fields. The flags and indexes are shared data, and the rules of Section 14.8 apply to them fully.
- **`while (!writeable) { … wait(); }`: `while`, not `if`.** When a waiting thread wakes up, the condition it was waiting for may no longer hold: another thread may have got in first and used up the free cell, and Java even permits *spurious wake-ups*, in which a thread returns from `wait()`

although nobody called `notify`. The only safe pattern is to test the condition again after every wake-up, which is what the `while` loop does [2], [11].

- **`wait()` releases the lock.** This is essential: while the producer waits inside `setSharedInt`, the consumer can enter `getSharedInt` on the same object, which it could not do if the producer still held the lock.
- **`notifyAll()` rather than `notify()`.** With exactly one producer and one consumer, `notify()` would work. With several producers and consumers, `notify()` might wake a thread of the wrong kind (a producer, when only a consumer could make progress), which goes back to waiting while the right thread sleeps on: the program stalls. `notifyAll()` wakes everyone and lets the `while` loops sort out who may proceed. It costs a few needless wake-ups; it never loses one.

The same exchange can be shown in a window instead of the console. **Figure 14.8** puts the lines of this very run into a Swing text area: the same messages, the same wrap-around, and the same total of 55.

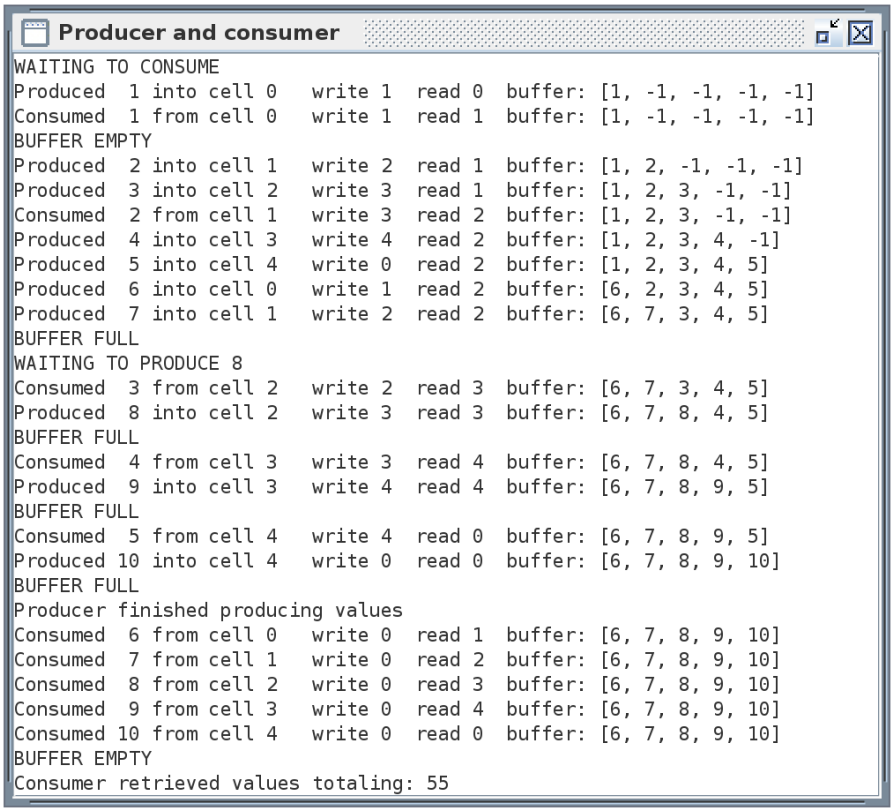


Figure 14.8 The output of `ProCon.java` shown in a Swing text area: the producer writes 1 to 10 into five circular cells, waits when the buffer is full, and the consumer's total is 55.

In Python the monitor is an explicit object, `threading.Condition`: a lock and a waiting room together. `with cond:` acquires its lock, `cond.wait()` releases the lock and waits, and `cond.notify_all()` wakes the waiters. **Listing 14.16** is a line-by-line translation.

Listing 14.16 `procon.py`: the same circular buffer with `threading.Condition`.

```
"""procon.py: one producer, one consumer and a five-cell circular buffer."""
import random
import threading
import time


class HoldIntegerSynchronized:
    def __init__(self):
        self.shared_int = [-1, -1, -1, -1, -1]
        self.writeable = True    # is there a free cell?
        self.readable = False    # is there a full cell?
        self.read_location = 0
        self.write_location = 0
        self.cond = threading.Condition()  # a lock plus a waiting room

    def _state(self):
        return (f"   write {self.write_location}  read {self.read_location}"
                f"  buffer: {self.shared_int}")

    def set_shared_int(self, value):
        with self.cond:
            while not self.writeable:      # while, not if: re-check after every wake-up
                print(f"WAITING TO PRODUCE {value}")
                self.cond.wait()           # releases the lock while waiting
            self.shared_int[self.write_location] = value
            self.readable = True
            cell = self.write_location
            self.write_location = (self.write_location + 1) % len(self.shared_int)
            print(f"Produced {value:2d} into cell {cell}" + self._state())
            if self.write_location == self.read_location:
                self.writeable = False
                print("BUFFER FULL")
            self.cond.notify_all()         # wake the consumer if it is waiting

    def get_shared_int(self):
        with self.cond:
            while not self.readable:
                print("WAITING TO CONSUME")
                self.cond.wait()
            self.writeable = True
            value = self.shared_int[self.read_location]
            cell = self.read_location
            self.read_location = (self.read_location + 1) % len(self.shared_int)
            print(f"Consumed {value:2d} from cell {cell}" + self._state())
            if self.read_location == self.write_location:
                self.readable = False
```

Listing 14.16 (Continued) *procon.py: the same circular buffer with threading.Condition.*

```
                print("BUFFER EMPTY")
            self.cond.notify_all()         # wake the producer if it is waiting
            return value


def main():
    buffer = HoldIntegerSynchronized()

    def produce():
        r = random.Random(7)
        for i in range(1, 11):
            time.sleep(r.randrange(200) / 1000)  # time to make the next value
            buffer.set_shared_int(i)
        print("Producer finished producing values")

    def consume():
        r = random.Random(31)
        total = 0
        for _ in range(10):
            time.sleep(r.randrange(700) / 1000)  # the consumer is slower
            total += buffer.get_shared_int()
        print(f"Consumer retrieved values totaling: {total}")

    producer = threading.Thread(target=produce)
    consumer = threading.Thread(target=consume)
    producer.start()
    consumer.start()
    producer.join()
    consumer.join()


if __name__ == "__main__":
    main()
```

```
WAITING TO CONSUME
Produced  1 into cell 0   write 1  read 0  buffer: [1, -1, -1, -1, -1]
Consumed  1 from cell 0   write 1  read 1  buffer: [1, -1, -1, -1, -1]
BUFFER EMPTY
Produced  2 into cell 1   write 2  read 1  buffer: [1, 2, -1, -1, -1]
Produced  3 into cell 2   write 3  read 1  buffer: [1, 2, 3, -1, -1]
Produced  4 into cell 3   write 4  read 1  buffer: [1, 2, 3, 4, -1]
Produced  5 into cell 4   write 0  read 1  buffer: [1, 2, 3, 4, 5]
Produced  6 into cell 0   write 1  read 1  buffer: [6, 2, 3, 4, 5]
BUFFER FULL
WAITING TO PRODUCE 7
Consumed  2 from cell 1   write 1  read 2  buffer: [6, 2, 3, 4, 5]
Produced  7 into cell 1   write 2  read 2  buffer: [6, 7, 3, 4, 5]
BUFFER FULL
WAITING TO PRODUCE 8
Consumed  3 from cell 2   write 2  read 3  buffer: [6, 7, 3, 4, 5]
Produced  8 into cell 2   write 3  read 3  buffer: [6, 7, 8, 4, 5]
BUFFER FULL
WAITING TO PRODUCE 9
Consumed  4 from cell 3   write 3  read 4  buffer: [6, 7, 8, 4, 5]
Produced  9 into cell 3   write 4  read 4  buffer: [6, 7, 8, 9, 5]
BUFFER FULL
Consumed  5 from cell 4   write 4  read 0  buffer: [6, 7, 8, 9, 5]
Produced 10 into cell 4   write 0  read 0  buffer: [6, 7, 8, 9, 10]
BUFFER FULL
Producer finished producing values
Consumed  6 from cell 0   write 0  read 1  buffer: [6, 7, 8, 9, 10]
Consumed  7 from cell 1   write 0  read 2  buffer: [6, 7, 8, 9, 10]
Consumed  8 from cell 2   write 0  read 3  buffer: [6, 7, 8, 9, 10]
Consumed  9 from cell 3   write 0  read 4  buffer: [6, 7, 8, 9, 10]
Consumed 10 from cell 4   write 0  read 0  buffer: [6, 7, 8, 9, 10]
BUFFER EMPTY
Consumer retrieved values totaling: 55
```

The story is the same, with different sleep times from Python's random number generator and seeds 7 and 31: the consumer waits for the first value, the producer races ahead and fills the buffer at 6, then waits three times, for 7, 8 and 9. The values, the order and the total of 55 are, as always, the same. Java offers the same explicit style too: a `ReentrantLock` can create `Condition` objects with `lock.newCondition()`, whose `await()` and `signalAll()` play the roles of `wait()` and `notifyAll()`, and a single lock may have several conditions, such as one for "not full" and one for "not empty".

14.10 Producer and Consumer the Easy Way: Blocking Queues

Writing `wait` and `notifyAll` correctly is a good exercise and a poor habit. The standard libraries of both languages contain buffers that already do all of it. In Java, a **`BlockingQueue`** has a method `put(x)`, which waits while the queue is full, and a method `take()`, which waits while it is empty; `ArrayBlockingQueue` has a fixed capacity and `LinkedBlockingQueue` may be unbounded [6]. **Listing 14.17** is the producer and consumer of **Listing 14.15** again, reduced to its essentials.

Listing 14.17 `QueueDemo.java`: the producer–consumer exchange with an `ArrayBlockingQueue` of capacity five and a poison pill.

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class QueueDemo {
    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<Integer> buffer = new ArrayBlockingQueue<>(5);  // five cells
        final int done = -1;  // a "poison pill": no real value is negative

        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    buffer.put(i);       // waits while the queue is full
                }
                buffer.put(done);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
        Thread consumer = new Thread(() -> {
            int sum = 0;
            int count = 0;
            try {
                while (true) {
                    int value = buffer.take();  // waits while the queue is empty
                    if (value == done) {
                        break;
                    }
                    sum += value;
                    count++;
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            System.out.println("Consumer took " + count + " values totaling: " + sum);
        });
        producer.start();
        consumer.start();
        producer.join();
        consumer.join();
        System.out.println("queue left empty: " + buffer.isEmpty());
    }
}
```

```
Consumer took 10 values totaling: 55
queue left empty: true
```

All the flags, indexes, `while` loops and notifications of **Listing 14.15** have disappeared into the queue, and the output is the same on every run. One new idea appears: the **poison pill**. How does the consumer know that no more values will come? The producer puts a special value, here -1, which no real value can equal, and the consumer stops when it takes it. With several consumers, the producer puts one pill per consumer.

The `catch (InterruptedException e)` blocks call `Thread.currentThread().interrupt()` rather than silently ignoring the exception. Catching `InterruptedException` clears the thread's interrupted flag; setting it again preserves the information for any code further up that wants to know whether the thread was asked to stop. It is the polite thing to do whenever you catch an interrupt that you do not fully handle yourself [2].

Listing 14.18 uses Python's `queue.Queue`, whose `put` and `get` wait in the same way when the queue is full or empty.

Listing 14.18 `queue_demo.py`: the same exchange with `queue.Queue(maxsize=5)` and `None` as the poison pill.

```
"""queue_demo.py: the producer and consumer again, with queue.Queue doing the waiting."""
import queue
import threading

DONE = None  # a "poison pill": no real value is None


def main():
    buffer = queue.Queue(maxsize=5)  # five cells

    def produce():
        for i in range(1, 11):
            buffer.put(i)      # waits while the queue is full
        buffer.put(DONE)

    def consume():
        total = count = 0
        while True:
            value = buffer.get()  # waits while the queue is empty
            if value is DONE:
                break
            total += value
            count += 1
        print(f"Consumer took {count} values totaling: {total}")

    producer = threading.Thread(target=produce)
    consumer = threading.Thread(target=consume)
    producer.start()
    consumer.start()
    producer.join()
    consumer.join()
    print("queue left empty:", buffer.empty())


if __name__ == "__main__":
    main()
```

```
Consumer took 10 values totaling: 55
queue left empty: True
```

`queue.Queue` is internally a list protected by a `threading.Lock` and two `threading.Condition` objects, one for "not empty" and one for "not full": exactly the machinery of **Listing 14.16**, written once, carefully, by the developers of the standard library. Use it. In both languages the blocking queue is the single most useful tool for concurrent programs, because it replaces *shared data* (which needs locks) with *messages* (which are handed from one thread to another and then belong to the receiver alone).

14.11 Deadlock

Locks cure races, and they bring a new danger of their own. A thread that holds one lock and waits for a second can wait forever if the second is held by a thread that is waiting for the first. Neither can move, and neither will ever let go. This is **deadlock**.

14.11.1 The dining philosophers

The most famous illustration was posed by Edsger Dijkstra in 1965 and given its present form by Tony Hoare [13]. Five philosophers sit at a round table, each with a plate of spaghetti. Between

each pair of neighbors lies one fork, so there are five forks in all, as **Figure 14.9** shows. A philosopher alternately thinks and eats. To eat, a philosopher needs *both* the fork on the left and the fork on the right; a fork can be held by only one philosopher at a time, so neighbors cannot eat at the same moment. After eating, a philosopher puts both forks down and thinks again. The spaghetti never runs out, and the philosophers never stop being hungry. The problem is to design a discipline of behavior, a *concurrent algorithm*, under which no philosopher starves, although no philosopher knows when the others want to eat or think.

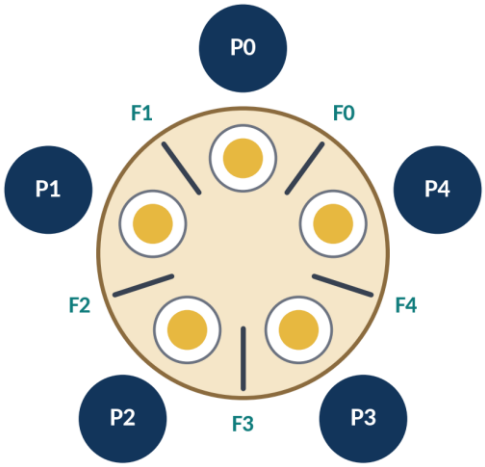


Figure 14.9 The dining philosophers: five philosophers, five plates of spaghetti, and one fork between each pair of neighbors. Philosopher P_i needs forks F_i and $F_{(i+1) \bmod 5}$, as in `Philosopher.java`.

The obvious discipline is: "pick up the left fork; then pick up the right fork; eat; put both down." Each philosopher is a thread and each fork is a lock. **Listing 14.19**, adapted from a well-known demonstration program [14], seats five such philosophers and draws the table every 100 ms. In each drawing, `^` is a fork lying on the table, `_0_` is a philosopher holding no fork, `<0` a philosopher holding only the left fork, and `<0>` a philosopher holding both and eating. Fork i is drawn just to the left of philosopher i, and philosopher i's right fork is fork $i + 1$, drawn to the left of the next philosopher (the table is round, so philosopher 5's right fork is fork 1, the first character of the line).

Listing 14.19 `Philosopher.java`: five philosophers, each a thread, each taking the left fork and then the right fork; `main` draws the table until every philosopher holds exactly one fork.

```
import java.util.Arrays;
import java.util.Random;

public class Philosopher extends Thread {
    static final int N = 5;                         // philosophers and forks
    static final int UNIT = 100;                    // one unit of time, in milliseconds
    static final int MAX_STEPS = 300;               // safety net: give up after 30 s
    static final Object[] forks = new Object[N];    // fork i lies to the left of
philosopher i
    static final String[] forkCell = new String[N];
    static final String[] dinerCell = new String[N];
    static final int[] meals = new int[N];
    static final Random random = new Random();

    private final int me;

    Philosopher(int me) {
        this.me = me;
    }

    // The picture of the table is shared, so it is changed and read under one lock.
    static synchronized void show(int fork, String forkText, int diner, String dinerText) {
        forkCell[fork] = forkText;
        dinerCell[diner] = dinerText;
    }

    static synchronized void ate(int diner) {
        meals[diner]++;
    }

    static synchronized String mealsEaten() {
        return Arrays.toString(meals);
    }

    static synchronized String table() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < N; i++) {
            sb.append(forkCell[i]).append(' ').append(dinerCell[i]).append(' ');
        }
        return sb.toString();
    }
```

Listing 14.19 (Continued) *Philosopher.java: five philosophers, each a thread, each taking the left fork and then the right fork; main draws the table until every philosopher holds exactly one fork.*

```
    @Override
    public void run() {
        int left = me;
        int right = (me + 1) % N;
        try {
            while (!isInterrupted()) {
                sleep(UNIT * random.nextInt(6));            // think
                synchronized (forks[left]) {                // take the left fork
                    show(left, " ", me, "<O ");
                    sleep(UNIT);
                    synchronized (forks[right]) {           // take the right fork
                        show(right, " ", me, "<O>");
                        sleep(UNIT);                        // eat
                        ate(me);
                        show(right, "^", me, "<O ");        // put the right fork back
                    }
                    show(left, "^", me, "_O_");             // put the left fork back
                }
            }
        } catch (InterruptedException e) {
            // interrupted while thinking or eating: leave the table
        }
    }

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < N; i++) {
            forks[i] = new Object();
            forkCell[i] = "^";
            dinerCell[i] = "_O_";
        }
        String locked = "  <O  ".repeat(N);  // every philosopher holds exactly one fork
        for (int i = 0; i < N; i++) {
            Thread t = new Philosopher(i);
            t.setDaemon(true);   // daemon threads cannot keep the program alive
            t.start();
        }
        System.out.println("The diner table:");
        for (int step = 1; step <= MAX_STEPS; step++) {
            String now = table();
            System.out.printf("%s %3d%n", now, step);
            if (now.equals(locked)) {
                Thread.sleep(5 * UNIT);  // make sure: nobody can move any more
                if (table().equals(locked)) {
                    System.out.println("The diner is locked.");
                    System.out.println("Meals eaten before the deadlock: " + mealsEaten());
                    return;
```

Listing 14.19 (Continued) *Philosopher.java: five philosophers, each a thread, each taking the left fork and then the right fork; main draws the table until every philosopher holds exactly one fork.*

```
                }
            }
            Thread.sleep(UNIT);
        }
        System.out.println("No deadlock within " + MAX_STEPS + " steps this time.");
    }
}
```

In one real run the table locked after 36 drawings:

```
The diner table:
^ _O_ ^ _O_   <O  ^ _O_ ^ _O_    1
^ _O_ ^ _O_   <O    <O    <O     2
  _O_ ^ _O_   <O    <O    <O>    3
  <O    <O    <O    <O>   _O_    4
  <O    <O    <O>   _O_ ^ _O_    5
  <O    <O>   _O_ ^ _O_ ^ _O_    6
  <O>   _O_   <O  ^ _O_ ^ _O_    7
  <O  ^ _O_   <O    <O    <O     8
  <O>   _O_   <O    <O    <O     9
  _O_ ^ _O_   <O    <O    <O>   10
  <O  ^ _O_   <O    <O>   _O_   11
  <O    <O    <O>   _O_ ^ _O_   12
  <O    <O>   _O_ ^ _O_ ^ _O_   13
  <O>   _O_   <O    <O  ^ _O_   14
^ _O_ ^ _O_   <O    <O    <O    15
  _O_ ^ _O_   <O    <O    <O>   16
  <O    <O    <O    <O>   _O_   17
  <O    <O    <O>   _O_ ^ _O_   18
  <O    <O>   _O_ ^ _O_ ^ _O_   19
  <O>   _O_   <O    <O  ^ _O_   20
^ _O_   <O    <O    <O>   _O_   21
  <O    <O    <O>   _O_   <O    22
  <O    <O>   _O_ ^ _O_   <O    23
  <O>   _O_ ^ _O_   <O    <O    24
  _O_ ^ _O_   <O    <O    <O>   25
  <O  ^ _O_   <O    <O>   _O_   26
  <O>   _O_   <O>   _O_ ^ _O_   27
^ _O_   <O  ^ _O_ ^ _O_ ^ _O_   28
^ _O_   <O>   _O_ ^ _O_   <O    29
  _O_ ^ _O_ ^ _O_   <O    <O>   30
^ _O_ ^ _O_ ^ _O_   <O>   _O_   31
  <O  ^ _O_ ^ _O_   <O  ^ _O_   32
  <O>   _O_   <O    <O    <O    33
  _O_ ^ _O_   <O    <O    <O>   34
  <O    <O    <O    <O>   _O_   35
  <O    <O    <O    <O    <O    36
The diner is locked.
Meals eaten before the deadlock: [7, 5, 5, 7, 6]
```

For a while all is well: philosophers pick up forks, some eat (`<O>`), and they put the forks down again. Then comes drawing 36. Every philosopher holds a left fork, no fork lies on the table, and every philosopher waits for a right fork that the neighbor holds. `main` checks for another half-second that nothing moves, then declares the diner locked; the five philosopher threads are daemons, so the program can end although they never will. Each of them managed a few meals first.

What varies: almost everything about the timeline. The philosophers think for a random 0 to 5 units, using an unseeded generator, so every run is different; in our runs the deadlock came after as few as 24 drawings and as many as 224. What does not vary: *eventually* all five pick up their left forks within the same short window, and from that moment no one ever eats again. The program has a safety net, `MAX_STEPS = 300`, in case a run is lucky for thirty whole seconds; none of ours was.

A few details of the code:

- `synchronized (forks[left]) { … synchronized (forks[right]) { … } }` is the "left, then right" discipline. The fork objects are plain `Object`s used only for their intrinsic locks.
- The drawing is shared data too: philosophers write it and `main` reads it. So `show`, `ate`, `table` and `mealsEaten` are `static synchronized`, which makes them all use one lock, the lock of the

`Philosopher` class. The drawing lock is always taken *inside* the fork locks and never the other way around; Section 14.12 explains why that consistency matters.

- `sleep(UNIT)` while holding the left fork is the dangerous gap: it stands for any work done between acquiring two locks, and the longer the gap, the sooner the deadlock comes.

Listing 14.20 is the same table in Python, with `threading.Lock` objects as forks and `with` blocks for "pick up ... put down".

Listing 14.20 `philosopher.py`: the dining philosophers in Python; each fork is a `threading.Lock`.

```
"""philosopher.py: five philosophers, five forks, and (sooner or later) a deadlock."""
import random
import threading
import time

N = 5            # philosophers and forks
UNIT = 0.1       # one unit of time, in seconds
MAX_STEPS = 300  # safety net: give up after 30 s
forks = [threading.Lock() for _ in range(N)]  # fork i lies to the left of philosopher i
fork_cell = ["^"] * N
diner_cell = ["_O_"] * N
meals = [0] * N
picture = threading.Lock()  # guards the three lists above


def show(fork, fork_text, diner, diner_text):
    with picture:
        fork_cell[fork] = fork_text
        diner_cell[diner] = diner_text


def ate(diner):
    with picture:
        meals[diner] += 1


def table():
    with picture:
        return "".join(f"{fork_cell[i]} {diner_cell[i]} " for i in range(N))


def philosopher(me):
    left, right = me, (me + 1) % N
    while True:
        time.sleep(UNIT * random.randrange(6))    # think
        with forks[left]:                         # take the left fork
            show(left, " ", me, "<O ")
            time.sleep(UNIT)
            with forks[right]:                    # take the right fork
                show(right, " ", me, "<O>")
                time.sleep(UNIT)                  # eat
                ate(me)
                show(right, "^", me, "<O ")       # put the right fork back
            show(left, "^", me, "_O_")            # put the left fork back
```

Listing 14.20 (Continued) *philosopher.py: the dining philosophers in Python; each fork is a threading.Lock.*

```
def main():
    locked = "  <O  " * N  # every philosopher holds exactly one fork
    for i in range(N):
        # daemon threads cannot keep the program alive
        threading.Thread(target=philosopher, args=(i,), daemon=True).start()
    print("The diner table:")
    for step in range(1, MAX_STEPS + 1):
        now = table()
        print(f"{now} {step:3d}")
        if now == locked:
            time.sleep(5 * UNIT)  # make sure: nobody can move any more
            if table() == locked:
                print("The diner is locked.")
                with picture:
                    print("Meals eaten before the deadlock:", meals)
                return
        time.sleep(UNIT)
    print(f"No deadlock within {MAX_STEPS} steps this time.")


if __name__ == "__main__":
    main()
```

```
The diner table:
^ _O_ ^ _O_ ^ _O_   <O  ^ _O_    1
^ _O_ ^ _O_ ^ _O_   <O    <O     2
  _O_ ^ _O_ ^ _O_   <O    <O>    3
^ _O_ ^ _O_ ^ _O_ ^ _O_ ^ _O_    4
  <O    <O    <O  ^ _O_ ^ _O_    5
  <O    <O    <O>   _O_   <O     6
  <O    <O>   _O_   <O    <O     7
  <O>   _O_ ^ _O_   <O    <O     8
  _O_ ^ _O_ ^ _O_   <O    <O>    9
^ _O_   <O  ^ _O_   <O>   _O_   10
^ _O_   <O    <O  ^ _O_   <O    11
  _O_   <O    <O>   _O_   <O>   12
^ _O_   <O>   _O_   <O  ^ _O_   13
  <O  ^ _O_ ^ _O_   <O>   _O_   14
  <O>   _O_   <O  ^ _O_   <O    15
  _O_   <O    <O>   _O_   <O>   16
^ _O_   <O>   _O_ ^ _O_ ^ _O_   17
  <O  ^ _O_   <O    <O  ^ _O_   18
  <O>   _O_   <O    <O>   _O_   19
^ _O_ ^ _O_   <O>   _O_ ^ _O_   20
^ _O_ ^ _O_ ^ _O_ ^ _O_ ^ _O_   21
^ _O_   <O  ^ _O_ ^ _O_   <O    22
  _O_   <O>   _O_   <O    <O>   23
^ _O_ ^ _O_   <O    <O>   _O_   24
  <O  ^ _O_   <O>   _O_   <O    25
  <O>   _O_ ^ _O_   <O    <O    26
  _O_ ^ _O_ ^ _O_   <O    <O>   27
^ _O_ ^ _O_ ^ _O_   <O>   _O_   28
^ _O_   <O  ^ _O_   <O    <O    29
  _O_   <O>   _O_   <O    <O>   30
^ _O_ ^ _O_   <O    <O>   _O_   31
  <O  ^ _O_   <O>   _O_ ^ _O_   32
  <O>   _O_   <O    <O  ^ _O_   33
^ _O_   <O    <O    <O    <O    34
  <O    <O    <O    <O    <O    35
The diner is locked.
Meals eaten before the deadlock: [5, 5, 6, 7, 7]
```

This run locked at drawing 35. Different runs lock at very different moments, from the tenth drawing to beyond the two hundred and fortieth in our trials, but every run we made locked in

the end. A Python thread stuck in `with forks[right]:` waits for the lock with no timeout, forever, exactly like a Java thread in `BLOCKED`.

14.11.2 Two clerks and two machines

The philosophers show deadlock arriving by bad luck. The laundry can show it arriving by design. Two clerks want to move a load from a washer to a dryer, so each needs *both* machines. Ani takes the washer first and then the dryer; Budi, working from the other end of the shop, takes the dryer first and then the washer. **Listing 14.21** lets them do so, and uses a `CountDownLatch` to make sure each has taken her or his first machine before either reaches for the second, so that the deadlock happens on every run.

Listing 14.21 `DeadlockDemo.java`: two threads take two locks in opposite orders; after one second the JVM's deadlock detector reports the cycle.

```
import java.lang.management.ManagementFactory;
import java.lang.management.ThreadInfo;
import java.lang.management.ThreadMXBean;
import java.util.concurrent.CountDownLatch;

public class DeadlockDemo {
    static final Object washer = new Object();
    static final Object dryer = new Object();
    static final CountDownLatch bothHoldOne = new CountDownLatch(2);

    // Takes the first machine, waits until the other clerk holds one too, then wants the
second.
    static void clerk(String firstName, Object first, String secondName, Object second) {
        String me = Thread.currentThread().getName();
        synchronized (first) {
            System.out.println(me + " holds the " + firstName);
            bothHoldOne.countDown();
            try {
                bothHoldOne.await();  // both clerks now hold one machine each
            } catch (InterruptedException e) {
                return;
            }
            System.out.println(me + " wants the " + secondName);
            synchronized (second) {
                System.out.println(me + " moved the load");  // never printed
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread ani = new Thread(() -> clerk("washer", washer, "dryer", dryer), "Ani");
        Thread budi = new Thread(() -> clerk("dryer", dryer, "washer", washer), "Budi");
        ani.setDaemon(true);   // so that the program can end although they never will
        budi.setDaemon(true);
        ani.start();
        budi.start();
        ani.join(1000);        // wait at most one second for each clerk
        budi.join(1000);
        System.out.println("after 1 s: Ani alive = " + ani.isAlive()
                + ", Budi alive = " + budi.isAlive());
```

Listing 14.21 (Continued) *DeadlockDemo.java: two threads take two locks in opposite orders; after one second the JVM's deadlock detector reports the cycle.*

```
        ThreadMXBean jvm = ManagementFactory.getThreadMXBean();
        long[] ids = jvm.findDeadlockedThreads();  // the JVM looks for a cycle
        if (ids != null) {
            for (ThreadInfo info : jvm.getThreadInfo(ids)) {
                System.out.println("deadlocked: " + info.getThreadName() + " is " +
info.getThreadState()
                        + ", waiting for a lock held by " + info.getLockOwnerName());
            }
        }
    }
}
```

```
Ani holds the washer
Budi holds the dryer
Ani wants the dryer
Budi wants the washer
after 1 s: Ani alive = true, Budi alive = true
deadlocked: Ani is BLOCKED, waiting for a lock held by Budi
deadlocked: Budi is BLOCKED, waiting for a lock held by Ani
```

The two `holds` lines may appear in either order, and so may the two `wants` lines; everything after them is the same on every run. A `CountDownLatch` is a one-shot gate: it starts with a count (here 2), `countDown()` lowers it, and `await()` waits until it reaches zero. After `bothHoldOne.await()` returns in both threads, each holds one machine and wants the other. **Figure 14.10** draws what happens next.

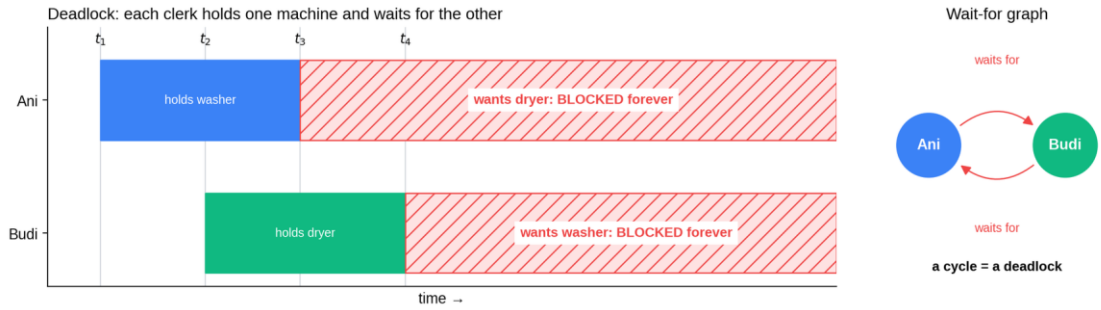


Figure 14.10 Timeline of the deadlock of **Listing 14.21**. At t_1 Ani takes the washer, at t_2 Budi takes the dryer, at t_3 and t_4 each asks for the other's machine and blocks forever. The wait-for graph on the right has a cycle, and a cycle in the wait-for graph is exactly a deadlock.

`main` cannot wait for them forever, so it joins each with a one-second timeout and then asks the JVM. `ManagementFactory.getThreadMXBean().findDeadlockedThreads()` builds the **wait-for graph** of the running program, with an arrow from each blocked thread to the thread that owns the lock it wants, and returns the threads that lie on a cycle. Here: Ani waits for Budi, and Budi waits for Ani. From outside a running program, the JDK command `jstack <pid>` prints every thread's stack and reports `Found one Java-level deadlock:` with the cycle.

> **DEFINITION.** A set of threads is **deadlocked** when each of them is waiting for a resource held by another thread in the set, so that none can ever proceed. Four conditions are necessary for a deadlock, and a deadlock is possible whenever all four hold at once [15]: (1) **mutual exclusion**: a resource can be held by only one thread; (2) **hold and wait**: a thread holds one resource while waiting for another; (3) **no preemption**: a resource cannot be taken away from its holder, only released by it; (4) **circular wait**: there is a cycle of threads, each waiting for a resource held by the next.

The four *Coffman conditions* are the key to prevention: break any one of them and deadlock becomes impossible. In the clerk example all four hold. A machine can serve one clerk at a time (1); each clerk holds one machine while waiting for the other (2); a clerk cannot snatch a machine from the other (3); and Ani waits for Budi who waits for Ani (4).

Listing 14.22 stages the same scene in Python. A `threading.Barrier(2)` plays the role of the latch: `wait()` blocks until two threads have arrived, then releases both.

Listing 14.22 `deadlock_demo.py`: the two clerks in Python; after one second both threads are still alive and both locks are still held.

```
"""deadlock_demo.py: two clerks, two machines, two locks taken in opposite orders."""
import threading

washer = threading.Lock()
dryer = threading.Lock()
both_hold_one = threading.Barrier(2)  # makes the unlucky timing certain
print_lock = threading.Lock()


def say(text):
    with print_lock:  # print() is not atomic: lines from two threads can mix
        print(text)


def clerk(first_name, first, second_name, second):
    me = threading.current_thread().name
    with first:
        say(f"{me} holds the {first_name}")
        both_hold_one.wait()          # both clerks now hold one machine each
        say(f"{me} wants the {second_name}")
        with second:
            say(f"{me} moved the load")  # never printed


def main():
    ani = threading.Thread(target=clerk, args=("washer", washer, "dryer", dryer),
                           name="Ani", daemon=True)
    budi = threading.Thread(target=clerk, args=("dryer", dryer, "washer", washer),
                            name="Budi", daemon=True)
    ani.start()
    budi.start()
    ani.join(1)   # wait at most one second for each clerk
    budi.join(1)
    print(f"after 1 s: Ani alive = {ani.is_alive()}, Budi alive = {budi.is_alive()}")
    print(f"washer locked = {washer.locked()}, dryer locked = {dryer.locked()}")
    # Daemon threads do not keep the program alive, so it ends here.


if __name__ == "__main__":
    main()
```

```
Ani holds the washer
Budi holds the dryer
Budi wants the washer
Ani wants the dryer
after 1 s: Ani alive = True, Budi alive = True
washer locked = True, dryer locked = True
```

Python has no built-in deadlock detector, but the symptoms are unmistakable: after a generous timeout both threads are alive and both locks are held. (`faulthandler.dump_traceback()` prints every thread's stack, showing where each is stuck.)

This listing also demonstrates a smaller concurrency lesson. Its first version used plain `print` calls from both threads, and one real run of that version printed:

```
Ani holds the washerBudi holds the dryer

Ani wants the dryer
Budi wants the washer
```

Python's `print` writes the text and the line ending as two separate operations, and another thread can print in between. The helper `say` wraps `print` in a lock so that each line comes out whole. Java's `System.out.println` is synchronized internally and never mixes two lines this way, although lines from different threads may of course still come out in any order.

14.12 Preventing Deadlock

A deadlock has four necessary conditions, so there are four places to attack it. In practice two of them are attacked most often: *circular wait*, by lock ordering, and *hold and wait* or *no preemption*, by giving up and retrying.

Lock ordering (Dijkstra's *resource hierarchy*). Number all the locks, and require every thread to acquire them in increasing order. Formally, if a thread holds lock ℓ_i and requests lock ℓ_j, then

$$i < j \tag{14.5}$$

must hold. Under Equation (14.5) a circular wait cannot exist: along any chain of waiting threads the lock numbers strictly increase, and a strictly increasing sequence can never come back to where it started. For the philosophers, the forks are numbered 0 to 4, and everyone picks up the lower-numbered fork first. Philosophers 0 to 3 still pick up left then right, but philosopher 4, whose forks are 4 and 0, picks up fork 0 first, the *right* one. That single change breaks the cycle. For the clerks, both would take the washer first.

Try, and back off (breaking *hold and wait*). Acquire the first lock; *try* to acquire the second with a time limit; if that fails, release the first, wait a random short time, and start over. A thread that has given its first lock back is no longer holding and waiting, so no cycle can persist. Java's `ReentrantLock.tryLock(time, unit)` and Python's `Lock.acquire(timeout=…)` support this directly. The random pause matters: without it, two threads that fail together retry together and can fail together forever, which is the livelock of Section 14.13.

An arbitrator (limiting the number of diners). Let at most $N - 1$ philosophers sit at the table at once, for instance with a **semaphore** holding $N - 1$ permits [16], [17]. With four philosophers and five forks, at least one seated philosopher can always get both forks. A waiter who hands out both forks at once, or none, is another form of arbitrator.

Chandy and Misra's solution lets philosophers request forks from each other by message; it works for any number of philosophers and resources, needs no central coordinator, and guarantees that nobody starves [18].

Listing 14.23 puts the naive discipline and three of these cures side by side. Each strategy seats five fresh philosophers with fresh forks, and each philosopher tries to eat three times.

Listing 14.23 `DeadlockFixes.java`: the left-then-right philosophers deadlock; lock ordering, `tryLock` with a timeout, and a four-seat semaphore all finish their fifteen meals.

```
import java.util.concurrent.Semaphore;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.ReentrantLock;

public class DeadlockFixes {
    static final int N = 5;
    static final int MEALS = 3;  // each philosopher eats three times, then leaves

    interface Diner {
        void dine(int me, ReentrantLock[] forks) throws InterruptedException;
    }

    static void pause(int maxMs) throws InterruptedException {
        Thread.sleep(ThreadLocalRandom.current().nextInt(maxMs + 1));
    }

    // Seats five philosophers with fresh forks, lets them eat, and reports what happened.
    static void run(String label, Diner diner) throws InterruptedException {
        ReentrantLock[] forks = new ReentrantLock[N];
        for (int i = 0; i < N; i++) {
            forks[i] = new ReentrantLock();
        }
        AtomicInteger meals = new AtomicInteger();
        Thread[] threads = new Thread[N];
        for (int i = 0; i < N; i++) {
            final int me = i;
            threads[i] = new Thread(() -> {
                try {
                    for (int m = 0; m < MEALS; m++) {
                        pause(20);              // think
                        diner.dine(me, forks);  // take forks, eat, put forks back
                        meals.incrementAndGet();
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
            threads[i].setDaemon(true);         // a deadlocked daemon cannot block the
exit
            threads[i].start();
        }
        boolean finished = true;
        for (Thread t : threads) {
            t.join(2000);                       // a deadlock shows up as a timeout
            finished &= !t.isAlive();
```

Listing 14.23 (Continued) *DeadlockFixes.java: the left-then-right philosophers deadlock; lock ordering, tryLock with a timeout, and a four-seat semaphore all finish their fifteen meals.*

```
        }
        System.out.printf("%-24s: %2d of %d meals, finished = %b%n",
                label, meals.get(), N * MEALS, finished);
    }

    // Takes two forks in the given order, eats, and puts them back.
    static void eatWith(ReentrantLock first, ReentrantLock second) throws
InterruptedException {
        first.lock();
        try {
            Thread.sleep(20);                   // the dangerous gap: holding one fork
            second.lock();
            try {
                pause(10);                      // eat
            } finally {
                second.unlock();
            }
        } finally {
            first.unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        // The problem: everybody takes the left fork first.
        run("0. left, then right", (me, forks) -> eatWith(forks[me], forks[(me + 1) % N]));

        // Fix 1: resource ordering. Always take the lower-numbered fork first.
        run("1. lock ordering", (me, forks) -> {
            int a = Math.min(me, (me + 1) % N);
            int b = Math.max(me, (me + 1) % N);
            eatWith(forks[a], forks[b]);
        });

        // Fix 2: tryLock with a timeout. Give the left fork back if the right one is busy.
        AtomicInteger retries = new AtomicInteger();
        run("2. tryLock with timeout", (me, forks) -> {
            ReentrantLock left = forks[me];
            ReentrantLock right = forks[(me + 1) % N];
            while (true) {
                left.lock();
                try {
                    pause(10);
                    if (right.tryLock(5, TimeUnit.MILLISECONDS)) {
                        try {
                            pause(10);          // eat
                            return;
                        } finally {
```

Listing 14.23 (Continued) *DeadlockFixes.java: the left-then-right philosophers deadlock; lock ordering, tryLock with a timeout, and a four-seat semaphore all finish their fifteen meals.*

```
                            right.unlock();
                        }
                    }
                } finally {
                    left.unlock();              // on failure: put the left fork down too
                }
                retries.incrementAndGet();
                pause(10);                      // back off for a random time, then retry
            }
        });
        System.out.println("   (tryLock gave up and retried " + retries.get() + " times)");

        // Fix 3: an arbitrator. At most N - 1 philosophers may sit at the table.
        Semaphore seats = new Semaphore(N - 1);
        run("3. at most four seated", (me, forks) -> {
            seats.acquire();
            try {
                eatWith(forks[me], forks[(me + 1) % N]);
            } finally {
                seats.release();
            }
        });
    }
}
```

```
0. left, then right     :  0 of 15 meals, finished = false
1. lock ordering        : 15 of 15 meals, finished = true
2. tryLock with timeout : 15 of 15 meals, finished = true
   (tryLock gave up and retried 4 times)
3. at most four seated  : 15 of 15 meals, finished = true
```

The naive version, `0. left, then right`, deadlocked before a single meal, and its threads are still stuck when the program ends (they are daemons, so they do not keep it alive). A 20-millisecond gap between the two forks makes all five pick up their left forks almost together; with a shorter gap it deadlocks less often, but in the end it always can. The three cures finished all fifteen meals. The number of `tryLock` retries varies from run to run (in our runs, anywhere from 2 to 7); the fifteen meals and `finished = true` do not.

Table 14.3 relates each cure to the condition it breaks.

Table 14.3 Ways to prevent deadlock, and the Coffman condition each one breaks.

Strategy	Coffman condition broken	Java	Python
lock ordering	circular wait	acquire in a fixed global order	the same
try with timeout, back off	hold and wait, no preemption	`tryLock(t, unit)`	`acquire(timeout=t)`
arbitrator (at most $N-1$)	circular wait (the cycle cannot close)	`Semaphore(N - 1)`	`threading.Semaphore(N - 1)`
one lock for everything	hold and wait (only one lock exists)	one `synchronized` object	one `Lock`
no shared resources	mutual exclusion	messages through a `BlockingQueue`	messages through `queue.Queue`

Listing 14.24 is the same experiment in Python.

Listing 14.24 `deadlock_fixes.py`: the naive philosophers and three cures in Python.

```
"""deadlock_fixes.py: the naive philosophers, and three ways to prevent the deadlock."""
import random
import threading
import time

N = 5
MEALS = 3  # each philosopher eats three times, then leaves


def pause(max_ms):
    time.sleep(random.randint(0, max_ms) / 1000)


def run(label, dine):
    """Seats five philosophers with fresh forks, lets them eat, and reports."""
    forks = [threading.Lock() for _ in range(N)]
    meals = [0]
    count_lock = threading.Lock()

    def philosopher(me):
        for _ in range(MEALS):
            pause(20)                # think
            dine(me, forks)          # take forks, eat, put forks back
            with count_lock:
                meals[0] += 1

    threads = [threading.Thread(target=philosopher, args=(i,), daemon=True)
               for i in range(N)]
    for t in threads:
        t.start()
    finished = True
    for t in threads:
        t.join(2)                    # a deadlock shows up as a timeout
        finished = finished and not t.is_alive()
    print(f"{label:<24}: {meals[0]:2d} of {N * MEALS} meals, finished = {finished}")


def eat_with(first, second):
    with first:
        time.sleep(0.02)             # the dangerous gap: holding one fork
        with second:
            pause(10)                # eat
```

Listing 14.24 (Continued) *deadlock_fixes.py: the naive philosophers and three cures in Python.*

```
def naive(me, forks):
    eat_with(forks[me], forks[(me + 1) % N])


def ordered(me, forks):
    a, b = sorted((me, (me + 1) % N))  # lower-numbered fork first
    eat_with(forks[a], forks[b])


retries = [0]
retry_lock = threading.Lock()


def polite(me, forks):
    left, right = forks[me], forks[(me + 1) % N]
    while True:
        with left:
            pause(10)
            if right.acquire(timeout=0.005):  # wait at most 5 ms
                try:
                    pause(10)       # eat
                    return
                finally:
                    right.release()
        # on failure: the with block has put the left fork down too
        with retry_lock:            # += on shared data needs a lock, too
            retries[0] += 1
        pause(10)                   # back off for a random time, then retry


seats = threading.Semaphore(N - 1)  # at most four philosophers at the table


def seated(me, forks):
    with seats:
        eat_with(forks[me], forks[(me + 1) % N])


def main():
    run("0. left, then right", naive)
    run("1. lock ordering", ordered)
    run("2. acquire with timeout", polite)
    print(f"   (acquire gave up and retried {retries[0]} times)")
    run("3. at most four seated", seated)
```

Listing 14.24 (Continued) *deadlock_fixes.py: the naive philosophers and three cures in Python.*

```
if __name__ == "__main__":
    main()
```

```
0. left, then right     :  0 of 15 meals, finished = False
1. lock ordering        : 15 of 15 meals, finished = True
2. acquire with timeout : 15 of 15 meals, finished = True
   (acquire gave up and retried 6 times)
3. at most four seated  : 15 of 15 meals, finished = True
```

The results agree with Java's. Two small points of the Python code deserve notice. `sorted((me, (me + 1) % N))` is lock ordering in one line. And the retry counter is incremented under its own `retry_lock`: `retries[0] += 1` is exactly the read-modify-write of Section 14.7, and five philosophers may retry at the same time.

> ✓ **KEY INSIGHT.** Lock ordering is the workhorse. It costs nothing at run time, needs no timeouts, and can be checked by reading the code: whenever a method holds one lock and asks for another, the second must come later in the agreed order. When the locks belong to objects of the same class (two wallets, two machines), give every object a unique number and always lock the smaller number first. Exercise 14.8 does exactly that.

14.13 Livelock and Starvation

Deadlocked threads do nothing. There are two other ways in which threads can fail to make progress while looking busy.

14.13.1 Livelock

You have probably met livelock in a corridor. You step to the left to let someone pass; at the same moment they politely step to *their* right, which puts them in front of you again. You both step the other way, and again, and again. Nobody is blocked, everybody is moving, and nobody gets anywhere.

> **DEFINITION.** A **livelock** is a situation in which threads keep changing state in response to each other, without blocking, and yet none of them makes progress.

Listing 14.25, adapted from a well-known example [19], builds the corridor out of two over-polite workers and one shared resource. Each worker wants to use the resource once. A worker uses the resource only when it is the owner; but if it is the owner and sees that the *other* worker is still active, it politely hands the resource over and waits for its turn. If both workers start active, each hands the resource to the other forever. To keep the demonstration finite, each worker also has a `politeness` limit and a `stopped` flag that `main` sets after 100 ms.

Listing 14.25 `Livelock.java`: two workers who always give way hand a resource back and forth without ever using it; limiting their politeness to one handover ends the livelock.

```
public class Livelock {
    static class CommonResource {
        private Worker owner;

        CommonResource(Worker d) {
            owner = d;
        }

        synchronized Worker getOwner() {
            return owner;
        }

        synchronized void setOwner(Worker d) {
            owner = d;
        }
    }

    static class Worker {
        private final String name;
        private final int politeness;     // how many times this worker will give way
        private volatile boolean active;  // still has work to do
        private volatile boolean stopped; // told to give up from outside
        private int handovers = 0;
        private int done = 0;

        Worker(String name, boolean active, int politeness) {
            this.name = name;
            this.active = active;
            this.politeness = politeness;
        }

        String getName() {
            return name;
        }

        boolean isActive() {
            return active;
        }
```

Listing 14.25 (Continued) *Livelock.java: two workers who always give way hand a resource back and forth without ever using it; limiting their politeness to one handover ends the livelock.*

```
        synchronized void work(CommonResource commonResource, Worker otherWorker) {
            while (active && !stopped) {
                // Wait for the resource to become available.
                if (commonResource.getOwner() != this) {
                    try {
                        wait(10);
                    } catch (InterruptedException e) {
                        return;
                    }
                    continue;
                }
                // If the other worker is also active, let it do its work first.
                if (otherWorker.isActive() && handovers < politeness) {
                    System.out.println(getName() + " : handover the resource to the worker
"
                            + otherWorker.getName());
                    handovers++;
                    commonResource.setOwner(otherWorker);
                    continue;
                }
                // Now use the common resource.
                System.out.println(getName() + " : working on the common resource");
                done++;
                active = false;
                commonResource.setOwner(otherWorker);
            }
        }

        synchronized String report() {
            return name + " handed over " + handovers + " times and did " + done + "
job(s)";
        }
    }

    static void scene(String title, int politeness) throws InterruptedException {
        System.out.println(title);
        Worker worker1 = new Worker("Worker 1", true, politeness);
        Worker worker2 = new Worker("Worker 2", true, politeness);
        CommonResource s = new CommonResource(worker1);
        Thread t1 = new Thread(() -> worker1.work(s, worker2));
        Thread t2 = new Thread(() -> worker2.work(s, worker1));
        t1.start();
        t2.start();
        Thread.sleep(100);            // give them 100 ms
        worker1.stopped = true;       // then tell both to give up
        worker2.stopped = true;
        t1.join();
        t2.join();
```

Listing 14.25 (Continued) *Livelock.java: two workers who always give way hand a resource back and forth without ever using it; limiting their politeness to one handover ends the livelock.*

```
        System.out.println("  " + worker1.report());
        System.out.println("  " + worker2.report());
    }

    public static void main(String[] args) throws InterruptedException {
        scene("Scene 1: both workers are always polite", Integer.MAX_VALUE);
        scene("Scene 2: each worker gives way at most once", 1);
    }
}
```

```
Scene 1: both workers are always polite
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
  Worker 1 handed over 8 times and did 0 job(s)
  Worker 2 handed over 8 times and did 0 job(s)
Scene 2: each worker gives way at most once
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : working on the common resource
Worker 2 : working on the common resource
  Worker 1 handed over 1 times and did 1 job(s)
  Worker 2 handed over 1 times and did 1 job(s)
```

In Scene 1, with unlimited politeness, the resource passed back and forth eight times each in 100 ms, and not one job was done; the exact number of handovers varies between runs (we saw 16 to 19 lines), and without the `stopped` flag it would grow for as long as the program ran. Notice that the threads were never `BLOCKED`: they were busy, waking every 10 ms, checking, handing over. That is the difference from deadlock, and it is why livelock is harder to spot: a monitoring tool sees threads that are alive and consuming processor time. In Scene 2 each worker gives way at most once, after which it uses the resource even if the other is active; both jobs are done after two handovers, and this scene prints the same lines on every run. **Figure 14.11** draws both scenes.

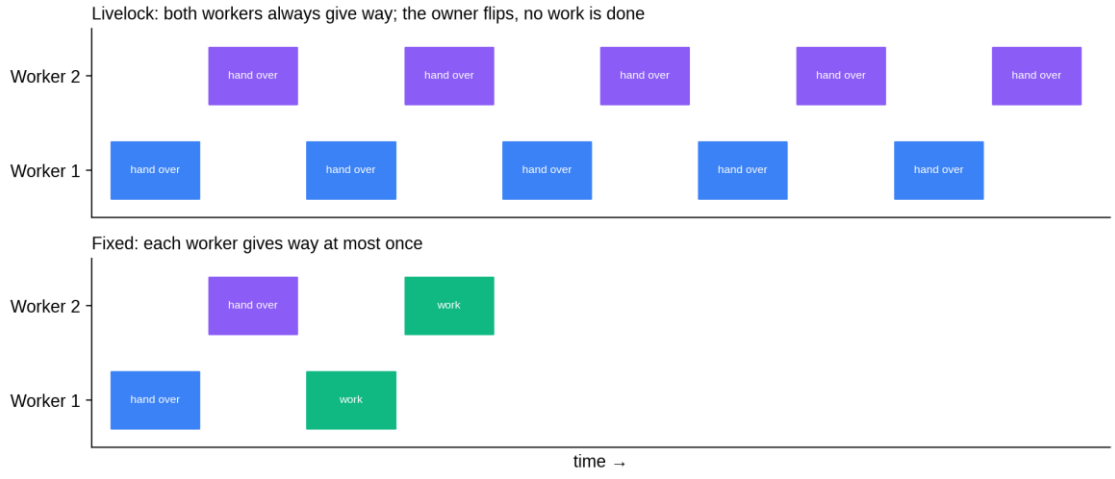


Figure 14.11 Livelock and its cure. Top: two always-polite workers hand the resource back and forth and never work. Bottom: each worker gives way at most once, so after two handovers both do their work.

Note that `active` and `stopped` are `volatile` (each worker reads the other's `active` flag, and `main` writes both `stopped` flags) and that `CommonResource` guards its `owner` with synchronized methods; `wait(10)` is simply a polite 10 ms pause that releases the worker's own lock. The program has no data race at all, and still it makes no progress: livelock is a failure of *design*, not of synchronization.

Livelocks are cured by breaking the symmetry: limit the politeness, as Scene 2 does; give the workers different priorities (for example, only the worker with the higher number gives way); or, as the `tryLock` philosophers of **Listing 14.23** did, back off for a *random* time, so that two threads that collided once are unlikely to collide again. The simplest cure of all is often to let one thread do the work sequentially: if the two jobs must use the same resource one after the other anyway, running them in two threads gains nothing. As with deadlock, no general rule prevents livelock; what helps is care whenever a thread changes shared state *in reaction to* what other threads are doing.

Listing 14.26 is the same corridor in Python.

Listing 14.26 `livelock.py`: the two over-polite workers in Python.

```
"""livelock.py: two over-polite workers keep handing a resource to each other."""
import sys
import threading
import time


class CommonResource:
    def __init__(self, owner):
        self._owner = owner
        self._lock = threading.Lock()

    def get_owner(self):
        with self._lock:
            return self._owner

    def set_owner(self, owner):
        with self._lock:
            self._owner = owner


class Worker:
    def __init__(self, name, active, politeness):
        self.name = name
        self.active = active          # still has work to do
        self.politeness = politeness  # how many times this worker will give way
        self.stopped = False          # told to give up from outside
        self.handovers = 0
        self.done = 0

    def work(self, common_resource, other_worker):
        while self.active and not self.stopped:
            # Wait for the resource to become available.
            if common_resource.get_owner() is not self:
                time.sleep(0.01)
                continue
            # If the other worker is also active, let it do its work first.
            if other_worker.active and self.handovers < self.politeness:
                print(f"{self.name} : handover the resource to the worker
{other_worker.name}")
                self.handovers += 1
                common_resource.set_owner(other_worker)
                continue
            # Now use the common resource.
            print(f"{self.name} : working on the common resource")
            self.done += 1
            self.active = False
            common_resource.set_owner(other_worker)
```

Listing 14.26 (Continued) *livelock.py: the two over-polite workers in Python.*

```
    def report(self):
        return f"{self.name} handed over {self.handovers} times and did {self.done} job(s)"


def scene(title, politeness):
    print(title)
    worker1 = Worker("Worker 1", True, politeness)
    worker2 = Worker("Worker 2", True, politeness)
    s = CommonResource(worker1)
    t1 = threading.Thread(target=worker1.work, args=(s, worker2))
    t2 = threading.Thread(target=worker2.work, args=(s, worker1))
    t1.start()
    t2.start()
    time.sleep(0.1)          # give them 100 ms
    worker1.stopped = True   # then tell both to give up
    worker2.stopped = True
    t1.join()
    t2.join()
    print("  " + worker1.report())
    print("  " + worker2.report())


def main():
    scene("Scene 1: both workers are always polite", sys.maxsize)
    scene("Scene 2: each worker gives way at most once", 1)


if __name__ == "__main__":
    main()
```

```
Scene 1: both workers are always polite
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
  Worker 1 handed over 10 times and did 0 job(s)
  Worker 2 handed over 10 times and did 0 job(s)
Scene 2: each worker gives way at most once
Worker 1 : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1 : working on the common resource
Worker 2 : working on the common resource
  Worker 1 handed over 1 times and did 1 job(s)
  Worker 2 handed over 1 times and did 1 job(s)
```

The Python workers poll with `time.sleep(0.01)` instead of `wait(10)`, and the owner is protected by a lock inside `CommonResource`. In this run they handed the resource over ten times each in 100 ms before `main` stopped them; the fixed scene is again identical on every run.

14.13.2 Starvation and fairness

Starvation is the fate of a thread that could make progress but never gets the chance, because other threads always get the resource first. A philosopher whose two neighbors are so greedy that one of them is always eating starves, although there is no deadlock: the table as a whole makes progress. A thread with low priority on a busy system can starve; so can a writer that waits for a lock that a stream of readers keeps taking.

The standard defense is **fairness**: serve waiting threads in the order in which they arrived. Java's `synchronized` makes no fairness promise at all: when a lock is released, any waiting thread may get it, including one that arrived last. `new ReentrantLock(true)` and `new Semaphore(n, true)` are *fair*: the longest-waiting thread goes next. Fair locks are slower, so use them where starvation is a real risk rather than by default. Python's `threading.Lock` makes no fairness promise either; a `queue.Queue`, on the other hand, hands out items in first-in, first-out order, which is one more reason to prefer queues. The Chandy–Misra solution of Section 14.12 is starvation-free by construction [18].

14.14 Profitina Laundry: Machines as Worker Threads

We can now build the shop of **Figure 14.1**. The requirements:

- The front desk receives the day's orders. Each order has a number, a customer, a service and a weight. The services and prices are those of Chapter 7: **Wash & Fold** (`WF`) at Rp7,000 per kilogram and **Wash, Dry & Iron** (`WDI`) at Rp10,000 per kilogram, both charged for at least 3 kg and rounded up to the next half kilogram.
- Three washing machines, `W1`, `W2` and `W3`, work at the same time. Each machine takes the next waiting order as soon as it is free, washes it (in our simulation, 20 ms per kilogram), and records the price with the cashier.
- There is one cashier's ledger. It must end the day with the exact revenue, the exact number of orders, and the number of orders washed by each machine, however the machines' work interleaves.
- At closing time every machine must switch itself off, and the program must end.

Every tool we need is already on the table: a **blocking queue** for the orders (Section 14.10), **worker threads** that loop on `take()` (Section 14.4), a **synchronized ledger** for the shared totals (Section 14.8), and **poison pills** so that the machines know when to stop. **Figure 14.12** is the design.

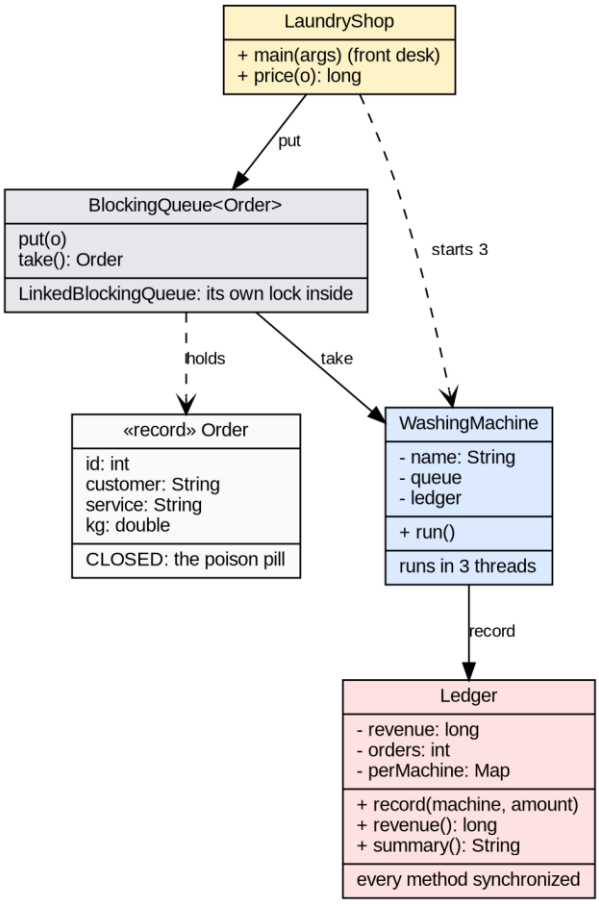


Figure 14.12 Class diagram of the concurrent laundry. The front desk (`main`) puts `Order` records into a `BlockingQueue`; three `WashingMachine` objects, each run by its own thread, take orders from it and record their prices in the single, fully synchronized `Ledger`.

Listing 14.27 is the program.

Listing 14.27 `LaundryShop.java`: three washing-machine threads take orders from a `LinkedBlockingQueue` and pay into a synchronized `Ledger`; poison pills switch them off.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.TreeMap;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class LaundryShop {
    record Order(int id, String customer, String service, double kg) { }

    static final Order CLOSED = new Order(0, "-", "-", 0);  // poison pill: "the shop is
closed"
    static final int MS_PER_KG = 20;                         // simulated washing time

    // Chapter 7's rule: at least 3 kg, rounded up to the next half kilogram.
    static long price(Order o) {
        long rate = o.service().equals("WF") ? 7000 : 10000;  // WF or WDI, per kg
        double chargeable = Math.ceil(2 * Math.max(o.kg(), 3.0)) / 2;
        return Math.round(rate * chargeable);
    }

    // The cashier: one ledger shared by every machine. One machine at a time may write.
    static class Ledger {
        private long revenue = 0;
        private int orders = 0;
        private final Map<String, Integer> perMachine = new TreeMap<>();

        synchronized void record(String machine, long amount) {
            revenue += amount;
            orders++;
            perMachine.merge(machine, 1, Integer::sum);
        }

        synchronized long revenue() {
            return revenue;
        }

        synchronized String summary() {
            return orders + " orders, revenue Rp" + String.format(Locale.US, "%,d",
revenue)
                    + ", per machine " + perMachine;
        }
    }
```

Listing 14.27 (Continued) *LaundryShop.java: three washing-machine threads take orders from a LinkedBlockingQueue and pay into a synchronized Ledger; poison pills switch them off.*

```
    static class WashingMachine implements Runnable {
        private final String name;
        private final BlockingQueue<Order> queue;
        private final Ledger ledger;

        WashingMachine(String name, BlockingQueue<Order> queue, Ledger ledger) {
            this.name = name;
            this.queue = queue;
            this.ledger = ledger;
        }

        @Override
        public void run() {
            try {
                while (true) {
                    Order o = queue.take();          // wait for the next order
                    if (o == CLOSED) {
                        break;
                    }
                    System.out.printf(Locale.US, "%s washes order %2d: %-7s %4.1f kg %s%n",
                            name, o.id(), o.customer(), o.kg(), o.service());
                    Thread.sleep((long) (o.kg() * MS_PER_KG));
                    ledger.record(name, price(o));   // pay at the one cashier
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            System.out.println(name + " switched off");
        }
    }

    public static void main(String[] args) throws InterruptedException {
        List<Order> today = List.of(
                new Order(1, "Ayu", "WF", 4.2), new Order(2, "Bagus", "WDI", 6.0),
                new Order(3, "Citra", "WF", 2.5), new Order(4, "Dewi", "WF", 7.8),
                new Order(5, "Eko", "WDI", 3.4), new Order(6, "Fitri", "WF", 5.1),
                new Order(7, "Gilang", "WDI", 8.0), new Order(8, "Hana", "WF", 3.0),
                new Order(9, "Indra", "WF", 6.6), new Order(10, "Joko", "WDI", 4.4),
                new Order(11, "Kartika", "WF", 9.3), new Order(12, "Lestari", "WDI", 2.0));
```

Listing 14.27 (Continued) *LaundryShop.java: three washing-machine threads take orders from a LinkedBlockingQueue and pay into a synchronized Ledger; poison pills switch them off.*

```
        BlockingQueue<Order> queue = new LinkedBlockingQueue<>();
        Ledger ledger = new Ledger();
        List<Thread> machines = new ArrayList<>();
        for (String name : new String[] {"W1", "W2", "W3"}) {
            Thread t = new Thread(new WashingMachine(name, queue, ledger), name);
            machines.add(t);
            t.start();
        }
        long t0 = System.nanoTime();
        for (Order o : today) {
            queue.put(o);                  // the front desk takes orders in
        }
        for (int i = 0; i < machines.size(); i++) {
            queue.put(CLOSED);             // one pill per machine
        }
        for (Thread t : machines) {
            t.join(10_000);                // never wait forever
        }
        long wallMs = (System.nanoTime() - t0) / 1_000_000;

        long expected = 0;
        double totalKg = 0;
        for (Order o : today) {
            expected += price(o);          // the same sum, computed by one thread
            totalKg += o.kg();
        }
        System.out.println("Ledger: " + ledger.summary());
        System.out.println("Ledger matches the one-thread total: " + (ledger.revenue() ==
expected));
        System.out.printf(Locale.US, "Washing time %d ms in total, done in about %d ms%n",
                Math.round(totalKg * MS_PER_KG), wallMs);
    }
}
```

One real run printed:

```
W1 washes order  1: Ayu      4.2 kg WF
W3 washes order  3: Citra    2.5 kg WF
W2 washes order  2: Bagus    6.0 kg WDI
W3 washes order  4: Dewi     7.8 kg WF
W1 washes order  5: Eko      3.4 kg WDI
W2 washes order  6: Fitri    5.1 kg WF
W1 washes order  7: Gilang   8.0 kg WDI
W3 washes order  8: Hana     3.0 kg WF
W2 washes order  9: Indra    6.6 kg WF
W3 washes order 10: Joko     4.4 kg WDI
W1 washes order 11: Kartika  9.3 kg WF
W2 washes order 12: Lestari  2.0 kg WDI
W3 switched off
W2 switched off
W1 switched off
Ledger: 12 orders, revenue Rp533,500, per machine {W1=4, W2=4, W3=4}
Ledger matches the one-thread total: true
Washing time 1246 ms in total, done in about 532 ms
```

and another, a moment later:

```
W1 washes order  2: Bagus    6.0 kg WDI
W3 washes order  1: Ayu      4.2 kg WF
W2 washes order  3: Citra    2.5 kg WF
W2 washes order  4: Dewi     7.8 kg WF
W3 washes order  5: Eko      3.4 kg WDI
W1 washes order  6: Fitri    5.1 kg WF
W3 washes order  7: Gilang   8.0 kg WDI
W2 washes order  8: Hana     3.0 kg WF
W1 washes order  9: Indra    6.6 kg WF
W2 washes order 10: Joko     4.4 kg WDI
W3 washes order 11: Kartika  9.3 kg WF
W2 washes order 12: Lestari  2.0 kg WDI
W1 switched off
W2 switched off
W3 switched off
Ledger: 12 orders, revenue Rp533,500, per machine {W1=3, W2=5, W3=4}
Ledger matches the one-thread total: true
Washing time 1246 ms in total, done in about 532 ms
```

Compare the two runs line by line, because they show precisely what a concurrent program may and may not change from one run to the next.

What varied. Which machine took which order: in the first run `W1` took order 1 and `W3` order 3; in the second, `W1` took order 2 and `W2` took both order 3 and order 4. The order of the first lines, which depends on which machine thread happened to reach `take()` first. The distribution of work: four orders per machine in the first run, but 3, 5 and 4 in the second. The order in which the machines switched off. And the elapsed time: 532 ms in both of these runs, but anywhere from 530 ms to 561 ms in six further runs, and different again on your computer.

What did not vary, and must never vary: all 12 orders were washed, each exactly once; the revenue was Rp533,500; the number of orders in the ledger was 12; and the ledger matched the total computed by one thread from the same list. You can check the revenue by hand with the rule of Chapter 7. Order 1 is 4.2 kg of Wash & Fold, charged as 4.5 kg, $4.5 \times 7000 = 31{,}500$; order 3 (2.5 kg) and order 8 (3.0 kg) are both charged as the 3 kg minimum, 21,000 each; order 12 is 2.0 kg of Wash, Dry & Iron, charged as 3 kg, 30,000; and so on through all twelve, which add up to 533,500.

Now the design decisions, one by one.

- `record Order(...)` is an immutable record: once the front desk has created an order, no thread can change it. Immutable objects can be passed between threads freely (Chapter 13); the queue hands each order to exactly one machine, and nobody needs to lock it.
- `CLOSED` is the poison pill, an ordinary `Order` object that the machines recognize by *identity* (`o == CLOSED`), not by its contents. `main` puts one pill per machine after the last real order. Because the queue is first-in, first-out, every real order is taken before any pill.
- `LinkedBlockingQueue` does all the waiting. A machine that calls `take()` on an empty queue waits (`WAITING` in **Figure 14.4**) and wakes up as soon as an order arrives. No machine can take an order that another machine has already taken: that guarantee is the queue's job, and its internal lock provides it.
- `Ledger` is the one cashier. All three of its methods are `synchronized`, so a machine recording a payment has the ledger to itself, and `summary()` and `revenue()` see every payment recorded before them. Remove the word `synchronized` from `record`, and three machines adding to `revenue` at the same time can lose payments exactly as **Listing 14.11** lost increments; Exercise 14.3 asks you to try.

- **The washing happens outside the lock.** `Thread.sleep` (the wash) is in `run()`, not inside `record()`. If a machine held the ledger's lock while washing, the other machines could not pay, and the three machines would effectively run one at a time. Keep critical sections short: do the work first, then lock, update, unlock.
- `t.join(10_000)` waits for each machine, but never forever. If a bug ever left a machine stuck, the program would still end, and the ledger's order count would reveal the problem.
- **The check** `ledger.revenue() == expected` compares the concurrent result with a sequential computation of the same thing, a simple and powerful way to test concurrent code.

How long *should* the day take? The machines wash 62.3 kg in total, which at 20 ms per kilogram is 1,246 ms of washing. With m machines and washing times $t_1, \ldots, t_k$, no schedule can finish sooner than

$$T_{\text{wall}} \geq \max\left(\frac{1}{m}\sum_{i=1}^{k} t_i\,,\ \max_i t_i\right) \tag{14.6}$$

because the three machines together cannot wash faster than three times one machine, and the longest single order cannot be split. Here Equation (14.6) gives max(1246/3,186) ≈ 415 ms. Our machines use the simple rule "take the next order when free", and tracing that rule by hand gives 498 ms (the long 9.3 kg order 11 starts late, at 312 ms, on the machine that becomes free first); the measured times of about 530 ms add the costs of starting threads and waking from `sleep`. Better schedules exist: a shop that puts its longest orders first would finish closer to the bound. That is a scheduling question, not a concurrency one, but it shows why "three machines" never quite means "three times as fast", in the spirit of Equation (14.1).

Listing 14.28 is the laundry in Python.

Listing 14.28 `laundry_shop.py`: the concurrent laundry in Python: `WashingMachine` subclasses `threading.Thread`, the ledger guards its totals with a `Lock`, and `None` is the poison pill.

```
"""laundry_shop.py: three washing machines take orders from one queue and pay one
cashier."""
import math
import queue
import threading
import time
from dataclasses import dataclass


@dataclass(frozen=True)
class Order:
    id: int
    customer: str
    service: str
    kg: float


CLOSED = None      # poison pill: "the shop is closed"
MS_PER_KG = 20     # simulated washing time
print_lock = threading.Lock()


def say(text):
    with print_lock:  # print() is not atomic: lines from two threads can mix
        print(text)


def price(o):
    """Chapter 7's rule: at least 3 kg, rounded up to the next half kilogram."""
    rate = 7000 if o.service == "WF" else 10000  # WF or WDI, per kg
    chargeable = math.ceil(2 * max(o.kg, 3.0)) / 2
    return round(rate * chargeable)


class Ledger:
    """The cashier: one ledger shared by every machine. One machine at a time may write."""

    def __init__(self):
        self._lock = threading.Lock()
        self._revenue = 0
        self._orders = 0
        self._per_machine = {}
```

Listing 14.28 (Continued) *laundry_shop.py: the concurrent laundry in Python: WashingMachine subclasses threading.Thread, the ledger guards its totals with a Lock, and None is the poison pill.*

```
    def record(self, machine, amount):
        with self._lock:
            self._revenue += amount
            self._orders += 1
            self._per_machine[machine] = self._per_machine.get(machine, 0) + 1

    def revenue(self):
        with self._lock:
            return self._revenue

    def summary(self):
        with self._lock:
            per = dict(sorted(self._per_machine.items()))
            return f"{self._orders} orders, revenue Rp{self._revenue:,}, per machine {per}"


class WashingMachine(threading.Thread):
    def __init__(self, name, orders, ledger):
        super().__init__(name=name)
        self.orders = orders
        self.ledger = ledger

    def run(self):
        while True:
            o = self.orders.get()           # wait for the next order
            if o is CLOSED:
                break
            say(f"{self.name} washes order {o.id:2d}: {o.customer:<7} {o.kg:4.1f} kg
{o.service}")
            time.sleep(o.kg * MS_PER_KG / 1000)
            self.ledger.record(self.name, price(o))  # pay at the one cashier
        say(f"{self.name} switched off")


def main():
    today = [
        Order(1, "Ayu", "WF", 4.2), Order(2, "Bagus", "WDI", 6.0),
        Order(3, "Citra", "WF", 2.5), Order(4, "Dewi", "WF", 7.8),
        Order(5, "Eko", "WDI", 3.4), Order(6, "Fitri", "WF", 5.1),
        Order(7, "Gilang", "WDI", 8.0), Order(8, "Hana", "WF", 3.0),
        Order(9, "Indra", "WF", 6.6), Order(10, "Joko", "WDI", 4.4),
        Order(11, "Kartika", "WF", 9.3), Order(12, "Lestari", "WDI", 2.0)]
```

Listing 14.28 (Continued) *laundry_shop.py: the concurrent laundry in Python: WashingMachine subclasses threading.Thread, the ledger guards its totals with a Lock, and None is the poison pill.*

```
    orders = queue.Queue()
    ledger = Ledger()
    machines = [WashingMachine(name, orders, ledger) for name in ("W1", "W2", "W3")]
    for m in machines:
        m.start()
    t0 = time.perf_counter()
    for o in today:
        orders.put(o)                       # the front desk takes orders in
    for _ in machines:
        orders.put(CLOSED)                  # one pill per machine
    for m in machines:
        m.join(10)                          # never wait forever
    wall_ms = (time.perf_counter() - t0) * 1000

    expected = sum(price(o) for o in today)  # the same sum, computed by one thread
    total_kg = sum(o.kg for o in today)
    print("Ledger:", ledger.summary())
    print("Ledger matches the one-thread total:", ledger.revenue() == expected)
    print(f"Washing time {round(total_kg * MS_PER_KG)} ms in total, done in about
{wall_ms:.0f} ms")


if __name__ == "__main__":
    main()
```

```
W3 washes order  1: Ayu      4.2 kg WF
W2 washes order  2: Bagus    6.0 kg WDI
W1 washes order  3: Citra    2.5 kg WF
W1 washes order  4: Dewi     7.8 kg WF
W3 washes order  5: Eko      3.4 kg WDI
W2 washes order  6: Fitri    5.1 kg WF
W3 washes order  7: Gilang   8.0 kg WDI
W1 washes order  8: Hana     3.0 kg WF
W2 washes order  9: Indra    6.6 kg WF
W1 washes order 10: Joko     4.4 kg WDI
W3 washes order 11: Kartika  9.3 kg WF
W2 washes order 12: Lestari  2.0 kg WDI
W1 switched off
W2 switched off
W3 switched off
Ledger: 12 orders, revenue Rp533,500, per machine {'W1': 4, 'W2': 4, 'W3': 4}
Ledger matches the one-thread total: True
Washing time 1246 ms in total, done in about 499 ms
```

The design is the same class for class. `@dataclass(frozen=True)` makes `Order` immutable, as Java's `record` does. `WashingMachine` subclasses `threading.Thread`, which suits a long-lived worker with its own name and its own references to the queue and the ledger. The ledger's three methods each hold `self._lock`. The machines print through `say()`, for the reason seen in Section 14.11.2. And because washing is *waiting* (`time.sleep`), the GIL is released during every wash, so the three Python machines overlap as well as the Java ones do: about 500 ms for 1,246 ms of washing. Had the machines been *computing* in pure Python instead, Section 14.6 tells us that the three threads would have taken turns, and a `ProcessPoolExecutor` would have been the right tool.

14.15 Profitina in Practice

Profitina is an AI visibility platform: it scans a brand's website and measures how ready the site is to be cited by AI answer engines, the discipline called *generative engine optimization* (GEO). Its

web application is written in Python with FastAPI and served by Uvicorn, and nearly every idea of this chapter appears in it in some form.

A request is a customer at the front desk. While one request waits for the database, others are served: I/O-bound concurrency, for which Section 14.6 showed that Python threads (and asynchronous tasks, their close relatives) work well.

The scheduler is a producer, the scanners are consumers. Profitina re-scans each brand automatically at a cadence set by its subscription tier. Deciding *what* is due and actually *doing* the scans are separated by a queue, in the shape of **Listing 14.27**: the scheduler produces jobs, workers consume them, and a slow site delays only the worker that is scanning it.

The local language model is the single cashier. Profitina runs its AI features on a local Qwen2.5 model through Ollama, so user data never leaves the server. One model can answer only so many requests at a time, so requests wait their turn in an orderly queue; and when no model is available, the application says so instead of pretending.

Counters that must not lose updates. A login rate limit counts failed attempts, and the PROFIT utility coin keeps balances with daily earning caps: shared counters updated by many concurrent requests, precisely the situation of **Listing 14.11**. Such values live in the PostgreSQL database, whose transactions and row locks play the role of `synchronized`. The lesson carries over exactly: every value that more than one request can change needs one guardian.

Tests as the gate. Races do not show up on demand, so a concurrent system relies on discipline: small critical sections, immutable messages, queues instead of shared lists, and a large automated test suite that must pass before every release. As always in Profitina, the software measures and explains; the user decides.

14.16 Common Mistakes, and How to Avoid Them

1. Calling `run()` instead of `start()`. The code runs, in the current thread, with no concurrency at all (**Listing 14.5**). *Avoid it:* `start()` creates the thread; `run()` is only what the thread will do. In Python, pass `target=f`, not `target=f()`.

2. Updating shared data without a lock. `count++`, `total += price`, `list.add(x)` on a shared `ArrayList`: each is a read-modify-write that loses updates under concurrency (**Listing 14.11**). *Avoid it:* for every field that more than one thread can change, name the lock that guards it, and hold that lock for *every* read and write.

3. Guarding the writes but not the reads. A reader without the lock may see a stale or half-updated value. *Avoid it:* make getters `synchronized` too (as in `Ledger`), or use `volatile`/atomic variables for single values.

4. Believing that `volatile` makes `++` atomic. It makes each read and each write visible; it does not join them (**Listing 14.11** uses a volatile field and still loses updates). *Avoid it:* use `AtomicInteger`, `LongAdder` or a lock for counters.

5. Believing that the GIL makes Python code thread-safe. The GIL protects the interpreter, not your invariants; a method of several bytecodes can be interrupted between any two (**Listing 14.12**). *Avoid it:* lock shared data in Python exactly as in Java.

6. Using `if` instead of `while` around `wait()`. A woken thread may find the condition false again, or may have woken spuriously. *Avoid it:* always `while (!condition) wait();` in Java, and `while not condition: cond.wait()` (or `cond.wait_for(predicate)`) in Python.

7. Calling `wait`/`notify` without holding the lock. Java throws `IllegalMonitorStateException`; Python raises `RuntimeError: cannot wait on un-acquired lock`. *Avoid it:* call them only inside `synchronized (obj)` on the same `obj`, or inside `with cond:`.

8. Acquiring locks in different orders. Two threads taking two locks in opposite orders will deadlock sooner or later (**Listing 14.21**). *Avoid it:* fix a global order (Equation (14.5)) and document it; lock by a unique number when the locks belong to objects of one class.

9. Holding a lock while doing slow work. Washing, sleeping, reading a file or calling a network service inside a critical section makes every other thread wait, and calling unknown code (a listener, an overridable method) while holding a lock invites deadlock. *Avoid it:* compute first, then lock, update and release.

10. Swallowing `InterruptedException`. An empty `catch` makes a thread impossible to stop. *Avoid it:* end the thread, or at least restore the flag with `Thread.currentThread().interrupt()`.

11. Waiting forever. `join()`, `take()` and `acquire()` without a limit hang the whole program if something goes wrong. *Avoid it:* in demonstrations and in production code that must stay responsive, use the versions with timeouts, and treat a timeout as a signal to investigate.

12. Printing from several Python threads without a lock. Lines can merge (Section 14.11.2). *Avoid it:* use a print lock, the `logging` module (which locks internally), or send messages through a queue to a single printing thread.

13. Trusting a concurrency test that passed. A race may appear once in a million runs (Equation (14.2)). *Avoid it:* prefer designs (immutability, queues, data confined to one thread) that make races impossible rather than unlikely.

14.17 Chapter Summary

- A **thread** is an independent sequence of instructions inside a process; threads have their own stacks but share the heap. **Concurrency** means several tasks in progress; **parallelism** means several executing at the same instant on different cores. The **scheduler** decides the interleaving, differently on every run.
- Create threads with a `Thread` subclass, a `Runnable` or lambda, or an `ExecutorService` (Java), and with a `threading.Thread` subclass, a target function, or a `ThreadPoolExecutor` (Python). Call `start()`, never `run()`; wait with `join()`. Stop threads cooperatively: `interrupt()` in Java, a `threading.Event` in Python, or a `volatile` flag.
- A Java thread is `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING` or `TERMINATED` (**Figure 14.4**); Python exposes `is_alive()`.
- CPython's **GIL** lets one thread execute Python bytecode at a time: threads help **I/O-bound** work, not **CPU-bound** pure-Python work, for which processes are used. Speedups are bounded by **Amdahl's law** (Equation (14.1)). Free-threaded Python (3.13 experimental, 3.14 supported) removes the GIL.
- A **race condition** makes the result depend on timing; the classic case is the **lost update** of a read-modify-write such as `c++` or `+=` (**Figure 14.6**). Of the $\binom{m+n}{m}$ interleavings (Equation (14.2)) only a few may be correct, so tests cannot prove a race absent.
- **Mutual exclusion** cures races: `synchronized` methods and blocks, `ReentrantLock`, `threading.Lock`; single variables can use **atomics** (`AtomicInteger`, `LongAdder`). A lock also guarantees **visibility**; `volatile` guarantees visibility only.

- Threads wait for conditions with `wait`/`notifyAll` or `Condition`, always in a `while` loop; a **monitor** is a lock plus a waiting room. `BlockingQueue` and `queue.Queue` implement the **producer–consumer** pattern for you; a **poison pill** ends the consumers.
- **Deadlock** requires mutual exclusion, hold and wait, no preemption and circular wait; a cycle in the **wait-for graph** is a deadlock. Prevent it by **lock ordering** (Equation (14.5)), **try with timeout and random back-off**, an **arbitrator** such as a semaphore, or by not sharing (**Table 14.3**).
- **Livelock** is activity without progress, cured by breaking symmetry (limits, priorities, random back-off). **Starvation** is a thread that never gets its turn, cured by **fairness**.
- In the laundry, three machine threads take orders from one queue and pay one synchronized ledger; which machine washes what varies between runs, the revenue of Rp533,500 never does.

14.18 Exercises

Exercise 14.1 (threads or not?). For each task, say whether running it in several threads would make it faster in Java, and in CPython with the GIL, and why: (a) downloading 50 web pages; (b) counting primes below ten million in pure code; (c) compressing 50 files with a compression library written in C; (d) keeping a Tkinter window responsive while a report is computed; (e) adding up a list of twelve prices.

Exercise 14.2 (counting interleavings). Two threads each execute `c++` once, as the three steps read, add, write. (a) Use Equation (14.2) to count the possible interleavings and list the ones that leave $c = 2$. (b) Three threads each execute `c++` once. How many interleavings are there, and how many are correct? (c) What does this say about testing?

Exercise 14.3 (break the cashier). Remove the word `synchronized` from `Ledger.record` in **Listing 14.27** and run the program twenty times. Report what you see, explain it, and explain why the program is nevertheless wrong. Name two things that could go wrong.

Exercise 14.4 (states). What does each `println` print, and why?

```
Object lock = new Object();
Thread t = new Thread(() -> {
    synchronized (lock) { }
});
System.out.println(t.getState());          // (1)
synchronized (lock) {
    t.start();
    Thread.sleep(200);
    System.out.println(t.getState());      // (2)
}
t.join();
System.out.println(t.getState());          // (3)
```

Exercise 14.5 (`if` instead of `while`). Suppose `getSharedInt` in **Listing 14.15** used `if (!readable) wait();` instead of `while`, and there were *two* consumer threads. Describe, step by step, an interleaving in which a consumer reads a value that is not there.

Exercise 14.6 (Amdahl). A report generator spends 80% of its time on work that splits perfectly across cores. Use Equation (14.1) to compute the best speedup on 2, 4 and 8 cores, and the limit as the number of cores grows without bound. How many cores give 90% of that limit?

Exercise 14.7 (why ordering works). Prove that the lock-ordering philosophers of **Listing 14.23** cannot deadlock, using Equation (14.5). Then show that it is enough for *one* philosopher to take the right fork first.

Exercise 14.8 (two wallets). Profitina Laundry sells prepaid wallets, and customers may transfer money to each other. `transfer(from, to, amount)` must lock both wallets. Write it in Java and Python so that one thread repeatedly transferring from Ayu to Budi and another from Budi to Ayu can never deadlock, and show that no money is created or destroyed.

Exercise 14.9 (express first). Extend the laundry so that express orders are washed before normal ones, and orders of the same kind in order of arrival. Use `PriorityBlockingQueue` in Java and `queue.PriorityQueue` in Python, with one machine, and show the order in which five orders (2 and 4 express) are washed.

Exercise 14.10 (a livelock in the cure). In the `tryLock` philosophers of **Listing 14.23**, a philosopher who fails to get the right fork puts down the left one and waits a *random* time before trying again. Describe what could happen if every philosopher waited exactly 10 ms instead, and why randomness fixes it.

Exercise 14.11 (deadlock or not?). For each situation, say whether a deadlock is possible, and name the Coffman condition that is missing if it is not. (a) Two threads, each using one lock, never holding two at once. (b) Thread 1 calls the synchronized method `a.f()`, which calls the synchronized method `b.g()`; thread 2 calls `b.h()`, which calls `a.k()`; all four methods are synchronized. (c) One Python thread executes `with lock:` twice, nested, on the same `threading.Lock`. (d) The same, but with a Java `synchronized (obj)` nested twice on the same object. (e) Thread A calls `b.join()` and thread B calls `a.join()`.

14.19 Solutions to the Exercises

Solution 14.1. (a) Faster in both: downloading is waiting for the network, and a waiting thread (in Python, one that has released the GIL) lets the others download. (b) Faster in Java with several cores; *not* faster in CPython with the GIL, because only one thread executes Python bytecode at a time; use processes in Python. (c) Faster in both: such libraries do their work in C and typically release the GIL while compressing. (d) Yes in both, but for responsiveness rather than speed: the report runs in a worker thread while the window's own thread keeps handling events (and only that thread may touch the widgets, Chapter 10). (e) No: twelve additions take far less time than creating a single thread.

Solution 14.2. (a) With $m = n = 3$, Equation (14.2) gives $\binom{6}{3} = 20$ interleavings. Writing A's steps as $a_1 a_2 a_3$ and B's as $b_1 b_2 b_3$, the result is 2 only if one thread's read comes after the other's write, that is, for $a_1 a_2 a_3 b_1 b_2 b_3$ and $b_1 b_2 b_3 a_1 a_2 a_3$: 2 correct, 18 lose an update and leave $c = 1$. (b) Three sequences of three steps can be merged in $\frac{9!}{3!\,3!\,3!} = 1680$ ways; only the $3! = 6$ in which the three increments run one after another without overlapping are correct. (c) Almost all interleavings are wrong, yet the scheduler usually picks harmless ones because each thread runs for a whole time slice at once; a test sees the rare bad interleaving only by luck. Tests can reveal races; they cannot prove their absence.

Solution 14.3. In our twenty runs of the modified program, all twenty printed `Ledger matches the one-thread total: true`. That is not evidence of correctness. Each machine calls `record` only four times, about 100 ms apart, and a call takes well under a microsecond, so the chance that two calls overlap is tiny, just as a race that loses one update in a million operations passes every short test. The program is wrong because two machines *can* call `record` at the same moment. Two things could then go wrong: (1) `revenue += amount` and `orders++` are read-modify-writes and can lose a payment or an order, exactly as in **Listing 14.11**; (2) `perMachine.merge` modifies a

`TreeMap`, which is not thread-safe at all; two concurrent modifications can corrupt its internal tree, lose entries, or throw `ConcurrentModificationException` in `summary()`. Also, without `synchronized` on `record`, the synchronized `summary()` no longer guarantees that it sees all the writes. To make the bug visible, make the machines record many payments in quick succession, for example a thousand per order with no sleep in between.

Solution 14.4. (1) `NEW`: the thread exists but has not been started. (2) `BLOCKED`: it has started and is trying to enter `synchronized (lock)`, which `main` holds for the whole 200 ms. (3) `TERMINATED`: once `main` leaves its block, `t` enters, finds an empty body, and ends; `join()` waits for that.

Solution 14.5. Let the buffer be empty and consumers C1 and C2 both waiting inside `getSharedInt` (each passed its `if`, found `readable` false, and called `wait()`). The producer stores one value, sets `readable = true`, and calls `notifyAll()`, waking both. C1 re-acquires the lock first, takes the value, finds the buffer empty and sets `readable = false`, then returns. C2 now re-acquires the lock. With `if`, it does *not* test `readable` again: it continues after `wait()`, "reads" the cell at `readLocation`, which holds an old, already consumed value, and moves `readLocation` past the producer's `writeLocation`. From then on the indexes are inconsistent and the buffer returns stale values forever. With `while`, C2 would test `readable`, find it false, and wait again.

Solution 14.6. With $f = 0.8$: $S(2) = 1/(0.2 + 0.4) = 1.67$; $S(4) = 1/(0.2 + 0.2) = 2.5$; $S(8) = 1/(0.2 + 0.1) = 3.33$; and $S(p) \to 1/0.2 = 5$ as $p \to \infty$. Ninety percent of the limit is 4.5: $1/(0.2 + 0.8/p) = 4.5$ gives $0.2 + 0.8/p = 0.2222$, so $0.8/p = 0.0222$ and $p = 36$. Thirty-six cores reach 4.5; the last half-unit of speedup would need infinitely many.

Solution 14.7. Number the forks 0 to 4. Under ordering, a philosopher holding a fork i requests only a fork j with $i < j$ (Equation (14.5)). Suppose there were a deadlock: a cycle of philosophers $P_1, \ldots, P_k$, each holding a fork and waiting for the fork held by the next, with P_k waiting for P_1's fork. Let x_t be the fork held by P_t. P_t waits for x_{t+1} while holding x_t, so $x_t < x_{t+1}$ for every t, and P_k waits for x_1 while holding x_k, so $x_k < x_1$. Then $x_1 < x_2 < \cdots < x_k < x_1$, which is impossible. Hence no deadlock. In the ordered version of **Listing 14.23**, philosophers 0 to 3 take $\min(i, i + 1) = i$ first, which is their left fork; only philosopher 4, with forks 4 and 0, takes fork 0 first, which is the right fork. So the rule "all left-first except one right-first" is exactly lock ordering, and one exception is enough to break the cycle.

Solution 14.8. Listing 14.29 gives every wallet a unique `id` and always locks the wallet with the smaller id first, whichever way the money flows, so both directions take the locks in the same order and Equation (14.5) holds.

Listing 14.29 `WalletTransfer.java`: transfers in both directions between two wallets, made deadlock-free by locking the smaller id first.

```
import java.util.concurrent.locks.ReentrantLock;

public class WalletTransfer {
    static class Wallet {
        final int id;                    // a unique number fixes the locking order
        final ReentrantLock lock = new ReentrantLock();
        private long balance;

        Wallet(int id, long balance) {
            this.id = id;
            this.balance = balance;
        }

        long balance() {
            lock.lock();
            try {
                return balance;
            } finally {
                lock.unlock();
            }
        }
    }

    // Always lock the wallet with the smaller id first, whichever way the money goes.
    static void transfer(Wallet from, Wallet to, long amount) {
        Wallet first = from.id < to.id ? from : to;
        Wallet second = from.id < to.id ? to : from;
        first.lock.lock();
        try {
            second.lock.lock();
            try {
                if (from.balance >= amount) {
                    from.balance -= amount;
                    to.balance += amount;
                }
            } finally {
                second.lock.unlock();
            }
        } finally {
            first.lock.unlock();
        }
    }
```

Listing 14.29 (Continued) *WalletTransfer.java: transfers in both directions between two wallets, made deadlock-free by locking the smaller id first.*

```
    public static void main(String[] args) throws InterruptedException {
        Wallet ayu = new Wallet(1, 500_000);
        Wallet budi = new Wallet(2, 500_000);
        final int times = 100_000;
        Thread one = new Thread(() -> {
            for (int i = 0; i < times; i++) {
                transfer(ayu, budi, 7_000);
            }
        });
        Thread two = new Thread(() -> {
            for (int i = 0; i < times; i++) {
                transfer(budi, ayu, 7_000);  // the opposite direction
            }
        });
        one.setDaemon(true);
        two.setDaemon(true);
        one.start();
        two.start();
        one.join(10_000);
        two.join(10_000);
        boolean finished = !one.isAlive() && !two.isAlive();
        System.out.println("finished without deadlock: " + finished);
        System.out.println("money in the system: Rp" + (ayu.balance() + budi.balance()));
    }
}
```

```
finished without deadlock: true
money in the system: Rp1000000
```

Each thread makes 100,000 transfers of Rp7,000. The balance check `from.balance >= amount` is inside both locks, so a wallet never goes negative, and the total stays exactly Rp1,000,000: every rupiah that leaves one wallet arrives in the other within the same critical section. If you replace `first`/`second` by `from`/`to`, the program deadlocks: in four of our five runs of that version, both threads were stuck and `finished without deadlock: false` appeared after the two ten-second joins. **Listing 14.30** is the Python version; `with first.lock, second.lock:` acquires the two locks in the written order.

Listing 14.30 `wallet_transfer.py`: the same deadlock-free transfers in Python.

```
"""wallet_transfer.py: transfers in both directions without deadlock, by lock ordering."""
import threading


class Wallet:
    def __init__(self, wallet_id, balance):
        self.id = wallet_id              # a unique number fixes the locking order
        self.lock = threading.Lock()
        self._balance = balance

    def balance(self):
        with self.lock:
            return self._balance


def transfer(source, target, amount):
    # Always lock the wallet with the smaller id first, whichever way the money goes.
    first, second = sorted((source, target), key=lambda w: w.id)
    with first.lock, second.lock:
        if source._balance >= amount:
            source._balance -= amount
            target._balance += amount


def main():
    ayu = Wallet(1, 500_000)
    budi = Wallet(2, 500_000)
    times = 100_000

    def ayu_pays():
        for _ in range(times):
            transfer(ayu, budi, 7_000)

    def budi_pays():
        for _ in range(times):
            transfer(budi, ayu, 7_000)   # the opposite direction
```

Listing 14.30 (Continued) *wallet_transfer.py: the same deadlock-free transfers in Python.*

```
    one = threading.Thread(target=ayu_pays, daemon=True)
    two = threading.Thread(target=budi_pays, daemon=True)
    one.start()
    two.start()
    one.join(10)
    two.join(10)
    finished = not one.is_alive() and not two.is_alive()
    print("finished without deadlock:", finished)
    print(f"money in the system: Rp{ayu.balance() + budi.balance()}")


if __name__ == "__main__":
    main()
```

```
finished without deadlock: True
money in the system: Rp1000000
```

Solution 14.9. A priority queue hands out the *smallest* element first. **Listing 14.31** makes `Order` comparable so that express orders are smaller than normal ones and, among equals, earlier ids are smaller. The poison pill has the largest possible id and is not express, so it sorts last.

Listing 14.31 `ExpressQueue.java`: a `PriorityBlockingQueue` lets express orders jump ahead of normal ones.

```java
import java.util.concurrent.PriorityBlockingQueue;

public class ExpressQueue {
    record Order(int id, String customer, boolean express) implements Comparable<Order> {
        @Override
        public int compareTo(Order other) {
            if (express != other.express) {
                return express ? -1 : 1;             // express orders first
            }
            return Integer.compare(id, other.id);    // then first come, first served
        }
    }

    static final Order CLOSED = new Order(Integer.MAX_VALUE, "-", false);  // sorts last

    public static void main(String[] args) throws InterruptedException {
        PriorityBlockingQueue<Order> queue = new PriorityBlockingQueue<>();
        queue.put(new Order(1, "Ayu", false));
        queue.put(new Order(2, "Bagus", true));
        queue.put(new Order(3, "Citra", false));
        queue.put(new Order(4, "Dewi", true));
        queue.put(new Order(5, "Eko", false));
        queue.put(CLOSED);
        Thread machine = new Thread(() -> {
            try {
                while (true) {
                    Order o = queue.take();
                    if (o == CLOSED) {
                        break;
                    }
                    System.out.println("W1 washes order " + o.id() + " (" + o.customer()
                            + (o.express() ? ", express)" : ")"));
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }, "W1");
        machine.start();
        machine.join();
    }
}
```

```
W1 washes order 2 (Bagus, express)
W1 washes order 4 (Dewi, express)
W1 washes order 1 (Ayu)
W1 washes order 3 (Citra)
W1 washes order 5 (Eko)
```

Because all orders are in the queue before the machine starts, the output is the same on every run. With orders arriving *while* the machine works, an express order can only overtake orders still waiting, never the one already in the machine. **Listing 14.32** uses tuples, which Python compares element by element: `(0, 2, "Bagus")` comes before `(1, 1, "Ayu")` because $0 < 1$.

Listing 14.32 `express_queue.py`: express orders first with `queue.PriorityQueue` and tuples.

```
"""express_queue.py: a priority queue lets express orders jump ahead."""
import queue
import threading


def main():
    orders = queue.PriorityQueue()
    # (0 = express, 1 = normal; then the order id) sorts express first, then by arrival.
    for order_id, customer, express in [(1, "Ayu", False), (2, "Bagus", True),
                                        (3, "Citra", False), (4, "Dewi", True),
                                        (5, "Eko", False)]:
        orders.put((0 if express else 1, order_id, customer))
    orders.put((2, 0, None))  # the closing pill sorts last

    def machine():
        while True:
            rank, order_id, customer = orders.get()
            if customer is None:
                break
            print(f"W1 washes order {order_id} ({customer}" + (", express)" if rank == 0
else ")"))

    t = threading.Thread(target=machine, name="W1")
    t.start()
    t.join()


if __name__ == "__main__":
    main()
```

```
W1 washes order 2 (Bagus, express)
W1 washes order 4 (Dewi, express)
W1 washes order 1 (Ayu)
W1 washes order 3 (Citra)
W1 washes order 5 (Eko)
```

Solution 14.10. Suppose all five philosophers pick up their left forks at the same moment. Each tries for 5 ms to get the right fork, fails, puts the left fork down, and waits exactly 10 ms. They all wake at the same moment, pick up their left forks together again, fail together again, and so on: a livelock. Nobody is blocked, the forks keep moving, and nobody eats. With a *random* back-off the philosophers wake at different moments; after a collision, the first to wake usually finds both forks free, eats, and the symmetry is broken. Network protocols such as Ethernet use the same idea, called randomized exponential back-off.

Solution 14.11. (a) Not possible: no thread holds one lock while waiting for another; *hold and wait* is missing. (b) Possible: thread 1 holds `a` and wants `b` while thread 2 holds `b` and wants `a`, a circular wait with all four conditions present. The danger is that the nesting is hidden inside method calls; this is why calling other objects' synchronized methods while holding a lock is risky. (c) A deadlock of one thread with itself: `threading.Lock` is not reentrant, so the inner `with lock:` waits forever for a lock its own thread holds. Use `threading.RLock`. (d) No deadlock: Java's intrinsic locks are reentrant, so the inner `synchronized` succeeds at once. (e) Deadlock: each thread waits for the other to terminate, and neither can; `join` behaves like a lock that is "held" by a thread until it ends. A `join` with a timeout would reveal it.

References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: Wiley, 2018.

[2] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Upper Saddle River, NJ, USA: Addison-Wesley, 2006.

[3] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*, ch. 17, "Threads and Locks." Oracle America, 2023. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se21/html/jls-17.html

[4] Python Software Foundation, *The Python Standard Library*, modules `threading`, `queue` and `concurrent.futures`, version 3.14. [Online]. Available: https://docs.python.org/3/library/threading.html

[5] J. Manson, W. Pugh, and S. V. Adve, "The Java memory model," in *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '05)*, Long Beach, CA, USA, 2005, pp. 378–391.

[6] Oracle, "Package java.util.concurrent," *Java SE 21 API Specification*, and "Lesson: Concurrency," *The Java Tutorials*. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/package-summary.html

[7] H. M. Deitel and P. J. Deitel, *Java: How to Program*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall, 2002.

[8] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. AFIPS Spring Joint Computer Conf.*, Atlantic City, NJ, USA, 1967, pp. 483–485.

[9] S. Gross, "PEP 703 – Making the Global Interpreter Lock optional in CPython," Python Enhancement Proposal 703, 2023. [Online]. Available: https://peps.python.org/pep-0703/

[10] T. Wouters, M. Page, and S. Gross, "PEP 779 – Criteria for supported status for free-threaded Python," Python Enhancement Proposal 779, 2025. [Online]. Available: https://peps.python.org/pep-0779/

[11] Oracle, "Synchronization" and "Guarded blocks," *The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

[12] C. A. R. Hoare, "Monitors: an operating system structuring concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[13] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, no. 2, pp. 115–138, 1971.

[14] H. Yang, "Deadlock sample program," *Java Tutorials: Herong's Tutorial Examples*. [Online]. Available: https://www.herongyang.com/

[15] E. G. Coffman, M. J. Elphick, and A. Shoshani, "System deadlocks," *ACM Computing Surveys*, vol. 3, no. 2, pp. 67–78, 1971.

[16] W. Stallings, *Operating Systems: Internals and Design Principles*, 9th ed. Hoboken, NJ, USA: Pearson, 2018.

[17] E. W. Dijkstra, "Cooperating sequential processes," Technological University, Eindhoven, The Netherlands, Tech. Rep. EWD-123, 1965; reprinted in F. Genuys, Ed., *Programming Languages*. London, U.K.: Academic Press, 1968, pp. 43–112.

[18] K. M. Chandy and J. Misra, "The drinking philosophers problem," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 4, pp. 632–646, 1984.

[19] LogicBig, "Java — Thread livelock." [Online]. Available: https://www.logicbig.com/

Chapter 15

Sockets, Networking, and the Team Project

By the end of this chapter you will be able to explain, in plain words and with a picture, how a message travels from one program to another across a network: what an IP address is, what a port is, what TCP and UDP promise, and why every conversation has a client and a server. You will open sockets in Java with `ServerSocket` and `Socket` and in Python with the `socket` module; write a line-based echo server and client and run them in two terminals; design a protocol on paper as a state machine before writing a line of it; serve many clients at once with the thread pools of Chapter 14; and build a small JSON-over-TCP order-status service for Profitina Laundry whose Java client talks happily to its Python server and whose Python client talks to its Java server, which we demonstrate by running both. You will speak HTTP from both languages to a server your own program starts and stops, protect the personal data that travels through your service in the spirit of Indonesia's personal data protection law, and test networked code automatically on ephemeral ports. The second half of the chapter turns you into a member of a team: how to choose a project, split it into parts with clear interfaces, work together with Git, write a README, test each other's parts, survive a server or a client that goes down, and present the result. A complete sample project brief and a checklist close the chapter.

15.1 Where We Left Off

Every program in this book so far has lived alone. It ran on one computer, read the keyboard or a file, and printed to the screen. Chapter 11 gave our programs windows and Chapter 14 gave them several threads, but even the busiest program of Chapter 14 never spoke to another program.

Real software is sociable. The cashier's laptop at Profitina Laundry asks a server whether order ORD-0713 is ready; the courier's phone tells the same server that a bag was delivered; the owner's dashboard asks it for today's takings. Each is a separate program, often written by a different person in a different language on a different machine, and yet they cooperate as if they were one. The glue between them is the network, and the programmer's handle on the network is the **socket**.

This chapter has two halves that belong together. The first teaches networking from zero: what happens between two computers, how Java and Python open a connection, how to agree on the words both sides will use, and how a server keeps many conversations going at once. The second teaches you to build such a system *with other people*, because a client–server application is the classic team project: one person writes the server, another the client, a third the interface, and they meet only at the interfaces they agreed on in advance. The shared lesson: whether the parties are programs or people, they succeed when they write down, before they start, exactly what each side will say and may expect.

> ✓ **KEY INSIGHT.** A network connection is just two streams, one in each direction, exactly like the file streams of Chapter 5. Everything hard about networking comes from three facts: the other side is a separate program you do not control, it may disappear at any moment, and both

sides must agree on the meaning of every byte. Objects help with all three: an object can hide the stream behind friendly methods, an exception can report the disappearance, and a protocol object can hold the agreement.

15.2 Networking from Zero

Before we write a line of code we need a small vocabulary, built with an analogy that fits the case study: a laundry business in a big building.

15.2.1 Hosts and IP addresses

A computer attached to a network is called a **host**: your laptop, the server in the back room of the laundry, your phone. To send anything to a host, the network needs its address, just as a courier needs the address of a building.

DEFINITION. An **IP address** (Internet Protocol address) is a number that identifies a host's connection to a network. An **IPv4** address is 32 bits long and is written as four decimal numbers from 0 to 255 separated by dots, such as `192.168.1.10`. An **IPv6** address is 128 bits long and is written as eight groups of hexadecimal digits separated by colons, such as `2001:db8::10` (a run of zero groups may be shortened to `::`).

A 32-bit number can take $2^{32} \approx 4.3$ billion values, far too few for a planet of phones; IPv6 is the long-term answer [1], [2]. Three kinds of address matter in this chapter:

- `127.0.0.1` (IPv4) and `::1` (IPv6) are the **loopback** address, "this very computer". A message sent there never leaves the machine. Every example in this chapter uses loopback, so it works without Wi-Fi and without exposing your program to strangers.
- `localhost` is a **host name** that the computer translates into the loopback address. Translating names into addresses is the job of the **Domain Name System** (DNS): for `app.profitina.com`, DNS answers with the server's address.
- Addresses such as `192.168.x.y` and `10.x.y.z` are **private**: used inside a home or office network and not reachable from the Internet directly. The laundry's server and the cashier's laptop would have addresses like these.

15.2.2 Ports

An address gets a message to the right building, but a building has many offices, and a computer runs many programs. The operating system needs to know which program a message is for. That is what a port is.

DEFINITION. A **port** is a 16-bit number that the operating system uses to deliver network data to the right program on a host. A server program asks the operating system for a port and *listens* on it; a client names that port when it connects.

Because a port has 16 bits, the number of possible ports on one host, for one protocol, is

$$2^{16} = 65\,536, \qquad \text{ports } 0,1,\ldots,65\,535. \tag{15.1}$$

Equation (15.1) is the whole supply, divided into three ranges [3]. Ports 0–1023 are **well-known**: 80 for HTTP, 443 for HTTPS, 22 for SSH, 53 for DNS; on Linux and macOS an ordinary user may not listen on them. Ports 1024–49151 are **registered** ports used by applications, such as 5432 for PostgreSQL; we use 5050, 5051 and 5052 for our three servers. Ports 49152–65535 are **ephemeral** ports that the operating system hands out automatically, for example to the client

side of every connection. When a server asks for port 0, the system picks a free ephemeral port for it, which is exactly what automated tests want: two tests can never fight over the same number.

A single conversation between two programs is identified by four numbers together, the **four-tuple**:

$$\text{connection} = (\, IP_c,\ p_c,\ IP_s,\ p_s \,) \tag{15.2}$$

where IP_c and p_c are the client's address and port and IP_s and p_s the server's. Equation (15.2) explains how a server listening on one port can talk to thousands of clients at once: two browser tabs on one laptop talking to the same server share three of the four numbers, but each tab gets its own ephemeral port p_c, so the conversations never get mixed up.

15.2.3 Layers: how a message travels

Sending a message across a network has several very different parts: turning bits into radio waves, finding a route, making sure nothing is lost, deciding what the bytes mean. Engineers keep them apart by stacking them in **layers**, each using only the service of the layer below. The Internet uses four [1]:

1. The **link layer** moves a frame of bits across one physical hop: one Wi-Fi radio to the access point, one Ethernet cable to the switch.
2. The **internet layer** (IP) moves a *packet* from the source host to the destination host across many hops, using the IP addresses. It does its best, but it promises nothing: packets may be lost, duplicated or arrive out of order.
3. The **transport layer** moves data from one *program* to another, using the ports. This is where TCP and UDP live, and where the choice between reliability and speed is made.
4. The **application layer** is where your program lives. It decides what the bytes mean: a line of text, a JSON object, an HTTP request.

Figure 15.1 follows one status request from the cashier's Java client to the laundry's Python server. On the way down, each layer wraps what it received in its own envelope (a *header*): TCP adds the ports, IP the addresses, the link layer the hardware addresses of the next hop. At the other end each layer removes its envelope and passes the contents up. The dotted lines show the most useful illusion in networking: each layer seems to talk directly to its twin on the other host. Our programs sit at the top; they write a line of JSON into a socket and trust the layers below to deliver it.

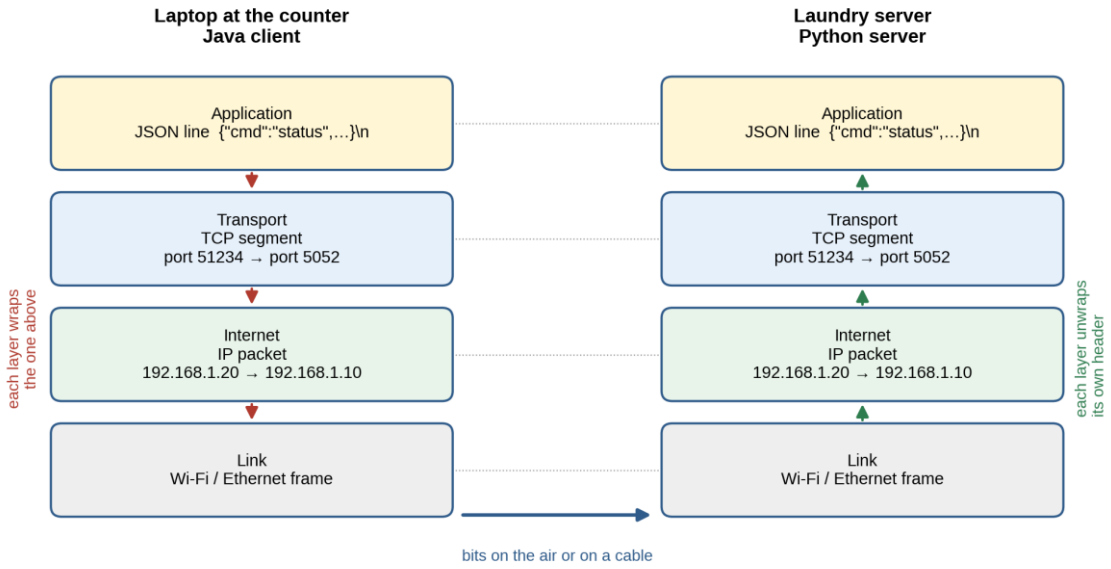


Figure 15.1 One request travelling through the four Internet layers. The client's layers wrap the JSON line in a TCP segment, an IP packet and a link-layer frame; the server's layers unwrap it in reverse. Each layer appears to talk directly to its twin on the other host.

15.2.4 TCP and UDP

The transport layer offers two services, and choosing between them is the first design decision of every networked program.

> **DEFINITION. TCP** (Transmission Control Protocol) provides a *connection*: a reliable, ordered, two-way *stream of bytes* between two programs. **UDP** (User Datagram Protocol) provides *datagrams*: independent messages, each delivered whole or not at all, with no promise of delivery, no promise of order, and no connection.

TCP is like a telephone call. Before anyone speaks, the two sides set up the call with the **three-way handshake** (SYN, SYN-ACK, ACK). From then on everything one side writes arrives at the other, in order, exactly once; lost packets are resent, and a slow receiver slows the sender down. The price is the setup time, some extra traffic, and one subtle point we meet in Section 15.4.4: TCP delivers a *stream* of bytes, so the boundaries between your messages are *not* preserved [4].

UDP is like sending postcards: no call to set up, each postcard arrives whole or not at all, possibly out of order. It is cheap, fast and keeps message boundaries, so it is used where a late message is worthless anyway: live voice and video, games, DNS lookups, and the "I am alive" heartbeats machines send every few seconds [5]. **Table 15.1** sums up.

Table 15.1 TCP and UDP compared.

Question	TCP	UDP
Connection set up first?	Yes, three-way handshake	No
Delivery guaranteed?	Yes; lost data is resent	No
Order guaranteed?	Yes	No
Message boundaries kept?	No: a byte stream	Yes: each datagram is whole
Speed and overhead	Slower to start, more traffic	Minimal
Java classes	`ServerSocket`, `Socket`	`DatagramSocket`, `DatagramPacket`
Python	`socket.SOCK_STREAM`	`socket.SOCK_DGRAM`
Typical uses	Web, e-mail, file transfer, our order service	Voice, video, games, DNS, heartbeats

The rest of this chapter uses TCP, because a laundry cannot afford to lose "order ORD-0713 was picked up". But UDP is so simple that it makes a perfect first program.

15.2.5 A first network program: datagrams to ourselves

Listing 15.1 sends three datagrams over loopback. The *receiver* socket is bound to port 0, so the operating system picks a free port; the *sender* fires three short status messages at it, and the receiver collects them.

Listing 15.1 `UdpHello.java`: three UDP datagrams sent over loopback from one socket to another.

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.nio.charset.StandardCharsets;

public class UdpHello {
    public static void main(String[] args) throws Exception {
        InetAddress loopback = InetAddress.getLoopbackAddress();  // 127.0.0.1
        try (DatagramSocket receiver = new DatagramSocket(0, loopback);  // port 0: OS
picks one
             DatagramSocket sender = new DatagramSocket()) {
            receiver.setSoTimeout(2000);  // never wait more than 2 s for a datagram
            int port = receiver.getLocalPort();
            System.out.println("receiver bound to " + loopback.getHostAddress()
                    + ", port chosen by the OS: " + (port > 0));

            String[] messages = {"machine W1: washing", "machine W2: idle", "machine D1:
drying"};
            for (String message : messages) {
                byte[] data = message.getBytes(StandardCharsets.UTF_8);
                sender.send(new DatagramPacket(data, data.length, loopback, port));
                System.out.println("sent     " + data.length + " bytes: " + message);
            }

            byte[] buffer = new byte[1500];
            for (int i = 0; i < messages.length; i++) {
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                receiver.receive(packet);  // one call, one whole datagram
                String text = new String(packet.getData(), 0, packet.getLength(),
                        StandardCharsets.UTF_8);
                System.out.println("received " + packet.getLength() + " bytes from "
                        + packet.getAddress().getHostAddress() + ": " + text);
            }
        }
    }
}
```

Compile and run it with `javac UdpHello.java` and `java UdpHello`:

```
receiver bound to 127.0.0.1, port chosen by the OS: true
sent     19 bytes: machine W1: washing
sent     16 bytes: machine W2: idle
sent     18 bytes: machine D1: drying
received 19 bytes from 127.0.0.1: machine W1: washing
received 16 bytes from 127.0.0.1: machine W2: idle
received 18 bytes from 127.0.0.1: machine D1: drying
```

Read it from the top.

- `InetAddress.getLoopbackAddress()` returns the loopback address as an `InetAddress`, Java's representation of an IP address.
- `new DatagramSocket(0, loopback)` creates a UDP socket and *binds* it to port 0 on loopback: "any free port, please". `getLocalPort()` reveals which port we got; we print only `port > 0`, because the number changes on every run.
- `new DatagramSocket()` creates the sender on some ephemeral port. Both sockets sit in the `try (...)` header, so they are closed automatically (Chapter 5): a socket is an operating-system resource, like an open file.
- `setSoTimeout(2000)` limits `receive` to two seconds; without it, a lost datagram would make `receive` wait forever.
- `getBytes(StandardCharsets.UTF_8)` turns text into bytes. Networks carry bytes, never characters, and this book always uses **UTF-8**, which represents every language's letters [6].

- `new DatagramPacket(data, data.length, loopback, port)` is a datagram with its destination; `send` hands it to the system. On the other side, `receive(packet)` blocks until one datagram arrives and fills the empty packet; `getLength()` and `getAddress()` tell how much arrived and from whom.

The output shows what distinguishes UDP: three `send` calls produced three `receive` results of exactly 19, 16 and 18 bytes. The boundaries survived. On loopback nothing was lost or reordered; across a real network, neither is promised.

Listing 15.2 is the same program in Python.

Listing 15.2 `udp_hello.py`: the same three datagrams in Python.

```
"""udp_hello.py: three datagrams sent over UDP to our own computer."""
import socket

LOOPBACK = "127.0.0.1"


def main():
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as receiver, \
         socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sender:
        receiver.bind((LOOPBACK, 0))  # port 0: the OS picks one
        receiver.settimeout(2.0)      # never wait more than 2 s for a datagram
        port = receiver.getsockname()[1]
        print(f"receiver bound to {LOOPBACK}, port chosen by the OS: {str(port >
0).lower()}")

        messages = ["machine W1: washing", "machine W2: idle", "machine D1: drying"]
        for message in messages:
            data = message.encode("utf-8")
            sender.sendto(data, (LOOPBACK, port))
            print(f"sent     {len(data)} bytes: {message}")

        for _ in messages:
            data, (host, _port) = receiver.recvfrom(1500)  # one call, one whole datagram
            print(f"received {len(data)} bytes from {host}: {data.decode('utf-8')}")


if __name__ == "__main__":
    main()
```

Run it with `python3 udp_hello.py` (on Windows, `python udp_hello.py` or `py udp_hello.py`). It prints exactly what the Java program printed:

```
receiver bound to 127.0.0.1, port chosen by the OS: true
sent     19 bytes: machine W1: washing
sent     16 bytes: machine W2: idle
sent     18 bytes: machine D1: drying
received 19 bytes from 127.0.0.1: machine W1: washing
received 16 bytes from 127.0.0.1: machine W2: idle
received 18 bytes from 127.0.0.1: machine D1: drying
```

Python's `socket` module is a thin wrapper around the operating system's own *Berkeley sockets* interface, part of Unix since 1983 [7], [8]. Java hides that interface behind a class per job; Python shows it almost unchanged:

- `socket.socket(socket.AF_INET, socket.SOCK_DGRAM)` creates a socket for IPv4 addresses (`AF_INET`) and datagrams (`SOCK_DGRAM`), that is, UDP.
- `bind((LOOPBACK, 0))` binds it; Python writes an address as a tuple `(host, port)`, and `getsockname()[1]` is the port obtained.

- `sendto` sends one datagram and `recvfrom(1500)` receives one with its sender's address; `with` closes both sockets; `str(port > 0).lower()` prints `true` so that the two outputs match line by line.

15.2.6 Clients and servers

Almost every networked program plays one of two roles.

> **DEFINITION.** A **server** is a program that starts first, binds a socket to a known port, and waits for others to contact it. A **client** is a program that knows the server's host and port and *initiates* the contact. The roles are about who starts the conversation, not about the size of the computer: a laptop can run a server, and a large machine can be a client.

Normally a server runs on a specific computer and has a socket bound to a specific port; it waits, listening for a client to make a connection request. The client knows the server's host name and port, and tries to meet the server there; it also binds to a local port of its own, usually assigned by the system [9]. **Figure 15.2** shows this moment.

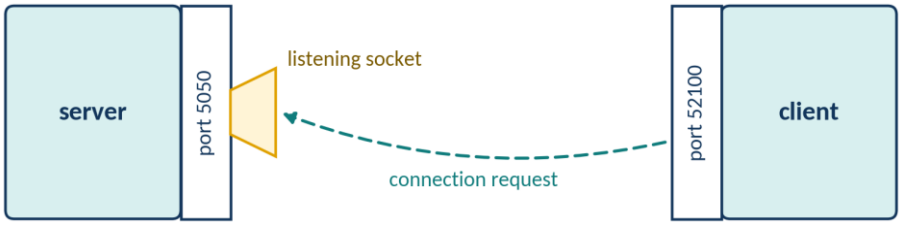


Figure 15.2 A client makes a connection request to the port on which the server is listening (here port 5050; the client's own port, such as 52100, is an ephemeral port chosen by its system).

If all goes well, the server *accepts* the connection and gets a **new** socket, bound to the same local port, whose remote end is the client's address and port. It needs a new socket so that it can keep listening on the original one while it tends to the client that just arrived. On the client side a socket is created, and the two programs now communicate by writing to and reading from their sockets (**Figure 15.3**).

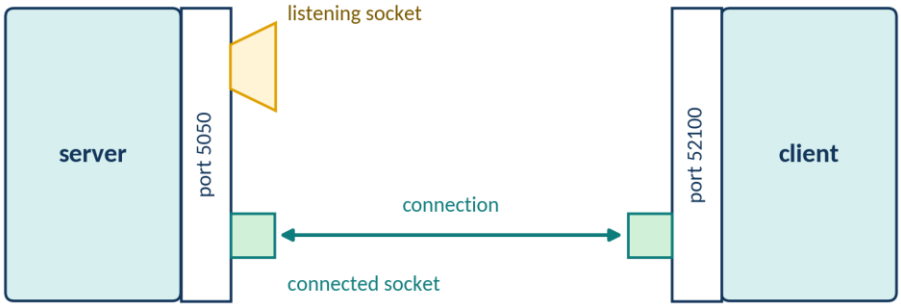


Figure 15.3 After acceptance the client and the server are connected through a new socket on the server's port 5050; the server's listening socket is still open for other clients.

This distinction is the most important idea for writing servers: a server has one **listening** socket for its whole life, which never carries data and only produces **connected** sockets, one per client, each identified by its own four-tuple (Equation (15.2)).

15.3 Sockets in Java and Python

> **DEFINITION.** A **socket** is one endpoint of a two-way communication link between two programs running on a network. A socket is bound to a port number so that the transport layer can identify the program that data is destined for [9].

Java's `java.net` package provides `Socket`, one side of a two-way connection, built on a platform-dependent implementation so that Java programs communicate in a platform-independent way, and `ServerSocket`, the listening socket that servers use to accept connections. To fetch a web page, the higher-level HTTP classes of Section 15.8 are more convenient; they use sockets underneath. Python's `socket.socket` plays both roles, depending on which methods you call. **Table 15.2** puts the two libraries side by side, in the order a program calls them, and **Figure 15.4** shows the calls on a timeline.

Table 15.2 The TCP socket calls in Java and Python, in the order a program uses them.

Step	Java	Python
Server: create, bind, listen	`new ServerSocket(port, backlog, addr)`	`s = socket.socket()`; `s.bind((host, port))`; `s.listen(backlog)`
Server: wait for a client	`Socket c = server.accept()`	`c, addr = s.accept()`
Client: connect	`new Socket(host, port)`	`socket.create_connection((host, port))`
Read text lines	`new BufferedReader(new InputStreamReader(c.getInputStream(), UTF_8))`	`c.makefile("r", encoding="utf-8")`
Write text lines	`new PrintWriter(new OutputStreamWriter(c.getOutputStream(), UTF_8), true)`	`c.makefile("w", encoding="utf-8")` then `flush()`
Raw bytes	`getInputStream().read(...)`, `getOutputStream().write(...)`	`c.recv(n)`, `c.sendall(data)`
Limit waiting	`c.setSoTimeout(ms)`	`c.settimeout(seconds)`
Close	`c.close()` or try-with-resources	`c.close()` or `with`
Other side closed	`readLine()` returns `null`	`readline()` returns `""`
Server not there	`ConnectException`	`ConnectionRefusedError`

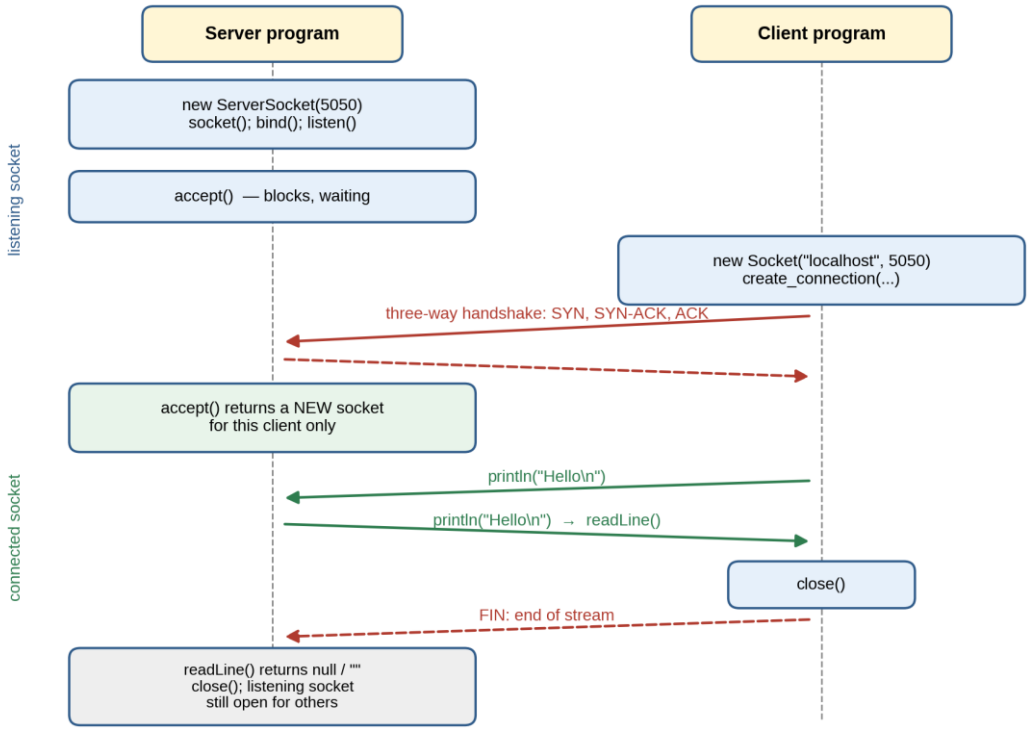


Figure 15.4 The life of one TCP connection. The server creates a listening socket and blocks in `accept()`; the client's connect triggers the three-way handshake; `accept()` returns a new connected socket; lines flow both ways; the client's `close()` sends FIN, which the server sees as the end of its input stream.

Three details deserve a comment before we use them.

Streams on top of sockets. A connected socket gives raw bytes: Java's `InputStream` and `OutputStream`, Python's `recv` and `sendall`. As in Chapter 5, we wrap them: in Java, `InputStreamReader` plus `BufferedReader` gives `readLine()`, and `OutputStreamWriter` plus `PrintWriter` gives `println`; in Python, `makefile` returns a file-like object with `readline`, `write` and iteration.

Always name the encoding. Otherwise both languages use the computer's default, which may differ between a Windows client and a Linux server, and the two will disagree about every letter beyond plain English.

Flushing. Writers buffer what you write, which is efficient for files and disastrous for conversations: the other side waits for a line still sitting in your buffer. Java's `PrintWriter(writer, true)` switches on *autoflush* at every `println`; in Python we call `writer.flush()`. Forgetting to flush is the commonest reason a first network program "hangs".

15.4 A Line-Based Echo Server and Client

Our first TCP pair is the simplest useful protocol: the client sends a line and the server sends it back, an *echo*. It is the "Hello, World" of networking, and echo servers are really used to check network paths and measure round-trip times.

15.4.1 The server

Listing 15.3 is the Java server. It takes the port number from the command line, waits for one client, echoes every line that client sends, and stops when the client hangs up.

Listing 15.3 `EchoServer.java`: accept one client and send every line back to it.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class EchoServer {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port>");
            System.exit(1);
        }
        int port = Integer.parseInt(args[0]);
        InetAddress loopback = InetAddress.getLoopbackAddress();

        // backlog 50: at most 50 connection requests may wait in the queue
        try (ServerSocket serverSocket = new ServerSocket(port, 50, loopback)) {
            System.out.println("EchoServer listening on " + loopback.getHostAddress()
                    + " port " + serverSocket.getLocalPort());
            try (Socket clientSocket = serverSocket.accept();  // blocks until a client
arrives
                 BufferedReader in = new BufferedReader(new InputStreamReader(
                         clientSocket.getInputStream(), StandardCharsets.UTF_8));
                 PrintWriter out = new PrintWriter(new OutputStreamWriter(
                         clientSocket.getOutputStream(), StandardCharsets.UTF_8), true)) {
                System.out.println("Client connected from "
                        + clientSocket.getInetAddress().getHostAddress());
                int lines = 0;
                String line;
                while ((line = in.readLine()) != null) {  // null: the client closed
                    out.println(line);                     // autoflush sends it at once
                    lines++;
                }
                System.out.println("Client closed the connection after " + lines + "
lines");
            }
        } catch (IOException e) {
            System.out.println("EchoServer error: " + e.getMessage());
        }
        System.out.println("EchoServer stopped");
    }
}
```

Line by line:

- `if (args.length != 1)` insists on exactly one argument, the port, and otherwise prints a usage message and exits with status 1.
- `new ServerSocket(port, 50, loopback)` creates the listening socket on loopback only, so only programs on this computer can connect, a safe default while developing (Section 15.9 explains how to open it). The **backlog**, 50, is how many connection requests the system may queue while the program is busy.
- `serverSocket.accept()` *blocks*: the thread stops here until a client connects. When one does, `accept` returns a new `Socket` for that client, the connected socket of **Figure 15.3**.
- The `try` header wraps the socket's streams in a UTF-8 `BufferedReader` and an autoflushing `PrintWriter`; try-with-resources closes all three.

- `while ((line = in.readLine()) != null)` is the heart of the server: `readLine` blocks until a whole line has arrived and returns `null` when the client closes its end. `out.println(line)` sends the line back at once.
- Any `IOException`, such as a port already in use, is reported in one line.

Listing 15.4 is the Python server.

Listing 15.4 `echo_server.py`: the echo server in Python.

```
"""echo_server.py: accept one client and send every line back to it."""
import socket
import sys

LOOPBACK = "127.0.0.1"


def main():
    if len(sys.argv) != 2:
        print("Usage: python3 echo_server.py <port>", file=sys.stderr)
        sys.exit(1)
    port = int(sys.argv[1])

    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
            if sys.platform != "win32":  # reuse the port at once after a restart
                server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            server_socket.bind((LOOPBACK, port))
            server_socket.listen(50)  # at most 50 connection requests may wait in the
queue
            print(f"EchoServer listening on {LOOPBACK} port
{server_socket.getsockname()[1]}",
                  flush=True)
            client_socket, (client_host, _) = server_socket.accept()  # blocks until a
client arrives
            with client_socket, \
                 client_socket.makefile("r", encoding="utf-8", newline="\n") as reader, \
                 client_socket.makefile("w", encoding="utf-8", newline="\n") as writer:
                print(f"Client connected from {client_host}", flush=True)
                lines = 0
                for line in reader:           # the loop ends when the client closes
                    writer.write(line)
                    writer.flush()            # send it at once
                    lines += 1
                print(f"Client closed the connection after {lines} lines")
    except OSError as e:
        print(f"EchoServer error: {e}")
    print("EchoServer stopped")


if __name__ == "__main__":
    main()
```

The calls follow the Python column of **Table 15.2**: `SOCK_STREAM` means a stream, that is, TCP; `bind`, `listen(50)` and `accept()` do what their names say. Two lines deserve comment.

- `setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)` lets the server bind again immediately after a restart. The side that closes a TCP connection first keeps a record of it for a minute or two (the TIME_WAIT state), and without this option Linux and macOS refuse to let a new server bind that port until the records expire. Java's `ServerSocket` sets the option by default; on Windows it would let two servers share a port, so we set it only elsewhere.
- `for line in reader:` reads the client's lines until it closes the connection; each line keeps its `\n`, so `writer.write(line)` returns it unchanged, and `flush()` sends it.

The early `print` calls pass `flush=True`: when output goes to a pipe rather than a terminal, Python buffers it, and a "listening" message could sit unseen while a test waits for it.

15.4.2 The client

Listing 15.5 is the client. It connects, then repeats: read a line from the keyboard, send it, read the echo, print it. It stops after the line `bye` or at the end of the input.

Listing 15.5 `EchoClient.java`: send lines typed by the user and print what comes back.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ConnectException;
import java.net.Socket;
import java.net.UnknownHostException;
import java.nio.charset.StandardCharsets;

public class EchoClient {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("Usage: java EchoClient <host> <port>");
            System.exit(1);
        }
        String host = args[0];
        int port = Integer.parseInt(args[1]);

        try (Socket socket = new Socket(host, port);  // 1. open a socket
             PrintWriter out = new PrintWriter(new OutputStreamWriter(      // 2. open
streams
                     socket.getOutputStream(), StandardCharsets.UTF_8), true);
             BufferedReader in = new BufferedReader(new InputStreamReader(
                     socket.getInputStream(), StandardCharsets.UTF_8));
             BufferedReader keyboard = new BufferedReader(new InputStreamReader(
                     System.in, StandardCharsets.UTF_8))) {
            System.out.println("Connected to " + host + " port " + port
                    + ". Type lines; 'bye' or end of input quits.");
            String userLine;
            while ((userLine = keyboard.readLine()) != null) {     // 3. talk
                out.println(userLine);
                String reply = in.readLine();
                if (reply == null) {
                    System.out.println("The server closed the connection.");
                    break;
                }
                System.out.println("echo: " + reply);
                if (userLine.equals("bye")) {
                    break;
                }
            }
        } catch (UnknownHostException e) {                 // 4.-5. try-with-resources
            System.out.println("Unknown host: " + host);   //       closes streams and
socket
        } catch (ConnectException e) {
            System.out.println("Cannot connect to " + host + " port " + port
                    + ": is the server running?");
```

Listing 15.5 (Continued) *EchoClient.java: send lines typed by the user and print what comes back.*

```
        } catch (IOException e) {
            System.out.println("I/O error: " + e.getMessage());
        }
    }
}
```

The numbered comments mark the five steps that every client follows, whatever its protocol [9]:

1. Open a socket: `new Socket(host, port)`. The constructor performs the connection: it looks up the host name, performs the three-way handshake and returns only when the connection exists. If nobody listens on that port, it throws `ConnectException`.
2. Open an input stream and an output stream on the socket.
3. Read from and write to the streams according to the server's protocol.
4. Close the streams.
5. Close the socket.

Only step 3 differs from client to client; with try-with-resources, steps 4 and 5 happen automatically, even when an exception interrupts step 3. The three `catch` clauses turn the three ways a connection can fail into sentences a user understands: an unknown host, a server that is not running (`ConnectException`, "connection refused"), and any other input or output error.

Listing 15.6 is the Python client.

Listing 15.6 `echo_client.py`: the echo client in Python.

```
"""echo_client.py: send lines typed by the user and print what comes back."""
import socket
import sys


def main():
    if len(sys.argv) != 3:
        print("Usage: python3 echo_client.py <host> <port>", file=sys.stderr)
        sys.exit(1)
    host, port = sys.argv[1], int(sys.argv[2])

    try:
        with socket.create_connection((host, port)) as sock, \
             sock.makefile("w", encoding="utf-8", newline="\n") as writer, \
             sock.makefile("r", encoding="utf-8", newline="\n") as reader:
            print(f"Connected to {host} port {port}. Type lines; 'bye' or end of input
quits.")
            for user_line in sys.stdin:
                user_line = user_line.rstrip("\r\n")
                writer.write(user_line + "\n")
                writer.flush()
                reply = reader.readline()
                if reply == "":
                    print("The server closed the connection.")
                    break
                print("echo: " + reply.rstrip("\n"))
                if user_line == "bye":
                    break
    except socket.gaierror:
        print(f"Unknown host: {host}")
    except ConnectionRefusedError:
        print(f"Cannot connect to {host} port {port}: is the server running?")
    except OSError as e:
        print(f"I/O error: {e}")


if __name__ == "__main__":
    main()
```

`socket.create_connection((host, port))` is Python's convenient version of step 1: it looks up the host name, tries each address the name resolves to (`localhost` may resolve to both `::1` and `127.0.0.1`), and returns a connected socket. The exceptions map as in **Table 15.2**: `socket.gaierror` for an unknown name ("get address info error") and `ConnectionRefusedError` when no server listens.

15.4.3 Running the pair in two terminals

A client–server program is two programs running at the same time. Open two terminals (in VS Code, **Terminal → New Terminal** twice), change into the chapter's `java` folder, compile once with `javac *.java`, and start the server first:

```
Terminal 1:  java EchoServer 5050
Terminal 2:  java EchoClient localhost 5050
```

Then type lines into terminal 2. For a reproducible run we fed the client the file `EchoClient.input.txt`, next to the program:

```
Hello, Profitina Laundry!
Is order ORD-0713 ready?
Terima kasih
bye
```

with `java EchoClient localhost 5050 < EchoClient.input.txt`. Terminal 1, the server, printed:

```
EchoServer listening on 127.0.0.1 port 5050
Client connected from 127.0.0.1
Client closed the connection after 4 lines
EchoServer stopped
```

and terminal 2, the client, printed:

```
Connected to localhost port 5050. Type lines; 'bye' or end of input quits.
echo: Hello, Profitina Laundry!
echo: Is order ORD-0713 ready?
echo: Terima kasih
echo: bye
```

When you type the lines yourself, each is followed by its echo. The Python pair (`python3 echo_server.py 5050` and `python3 echo_client.py localhost 5050 < echo_client.input.txt`) prints the same lines. And because both pairs speak the same protocol, UTF-8 text with one line per message, the Java client works with the Python server and vice versa; we tried it, and the output was the client output above, line for line. This is our first taste of *interoperability*: the protocol, not the language, decides who can talk to whom.

If no server is running, both clients say so in plain words:

```
Cannot connect to localhost port 5050: is the server running?
```

15.4.4 Lines, flushing and framing

TCP delivers a stream of bytes. If the client writes `Hello\n` and then `bye\n`, the server may receive `Hello\nbye\n` in one piece, or `Hel`, `lo\nb`, `ye\n` in three. TCP guarantees the bytes and their order, never the pieces, so "one `recv` returns one message" works on your laptop and fails on a busy network.

> **DEFINITION. Framing** is the rule a protocol uses to find where one message ends and the next begins inside a byte stream. The three common rules are: a **delimiter** (every message ends with a newline), a **length prefix** (every message starts with its length in bytes), and a **fixed size** (every message has the same length).

Every protocol in this chapter uses the newline delimiter, the easiest rule to debug because you can read the traffic yourself; `readLine()` and `readline()` implement it, however many pieces the bytes arrived in. Its one obligation is that a message never contains a raw newline, which our JSON messages of Section 15.7 satisfy automatically, since JSON writes a newline inside a string as `\n`.

Framing and flushing explain the two classic ways a first network program freezes. If the client sends `Hello` *without* a newline, the server's `readLine` waits for the end of the line while the client waits for the echo: a **deadlock**, as in Chapter 14, spread across two programs. If the client sends the newline but forgets to flush, the line stays in its buffer, with the same result. Exercise 15.3 causes both on purpose.

15.4.5 When the other side is not there

In a program that runs alone, a call returns or throws at once. Across a network it can also *hang*, because the other side may be slow, dead or cut off, and nothing says so. Three situations cover almost everything:

- **The server is down when the client connects.** Its machine refuses the handshake and the client gets `ConnectException` or `ConnectionRefusedError` at once (on Windows, after about

two seconds). If the whole machine is unreachable there is no refusal, and the connect waits, which is why Section 15.7 gives the connect a timeout.

- **The client disappears in the middle.** A normal close makes the server's read return `null` or `""`. A crashed computer or a pulled cable may produce nothing at all; only a **timeout** (`setSoTimeout`, `settimeout`) turns "waiting forever" into an exception.
- **The server disappears in the middle.** The client's next read returns end-of-stream or throws "connection reset"; the client must report it and must not pretend the last request succeeded.

> ⚠ **WARNING.** Every blocking network call needs an answer to the question "and what if the other side never answers?". In this chapter the answer is always a timeout plus an exception handler. A server without timeouts can be brought to a halt by clients that connect and then say nothing; Section 15.9 returns to this.

15.5 Designing a Protocol on Paper First

The echo protocol has one rule. Real protocols have many: who speaks first, what each side may say at each moment, what happens after a wrong message, how the conversation ends. When two people write the two sides, as they always do in a team project, the rules must be written down *before* either writes code; otherwise the two ideas of the protocol meet for the first time on demonstration day.

> **DEFINITION.** A **protocol** is the complete set of rules for a conversation between two programs: the messages that exist, their exact format, their order, the meaning of each, and what each side does when it receives a message it did not expect.

The best tool for writing a protocol on paper is the **state machine** that Chapter 1 used for the life of a laundry order: a small set of *states*, and for each the allowed messages and where they lead. We practice on the knock-knock joke, made famous by the Java tutorial [9], in which the server always speaks first:

```
Server: Knock! Knock!
Client: Who's there?
Server: Lettuce
Client: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
Client: n
Server: Bye.
```

Table 15.3 writes the rules as a table. Every row also says what happens when the client says something else; programmers forget those rows, and they are where a network program spends much of its real life.

Table 15.3 The knock-knock protocol as a table of states and messages. Matching ignores upper and lower case.

Server state	Client says	Server answers	Next state
`WAITING`	(connects)	`Knock! Knock!`	`SENT_KNOCK_KNOCK`
`SENT_KNOCK_KNOCK`	`Who's there?`	the clue, e.g. `Lettuce`	`SENT_CLUE`
`SENT_KNOCK_KNOCK`	anything else	`You're supposed to say "Who's there?"! …`	`SENT_KNOCK_KNOCK`
`SENT_CLUE`	`<clue> who?`	the answer + `Want another? (y/n)`	`ANOTHER`
`SENT_CLUE`	anything else	`You're supposed to say "<clue> who?"! …`	`SENT_KNOCK_KNOCK`
`ANOTHER`	`y`	`Knock! Knock!` (next joke)	`SENT_KNOCK_KNOCK`
`ANOTHER`	anything else	`Bye.`, then the server closes	`WAITING`

Figure 15.5 draws the same rules as a state diagram, which doubles as the protocol's flowchart. The table is better for checking that every case is covered, the diagram for seeing the shape of the conversation; a team should keep both in the repository next to the code.

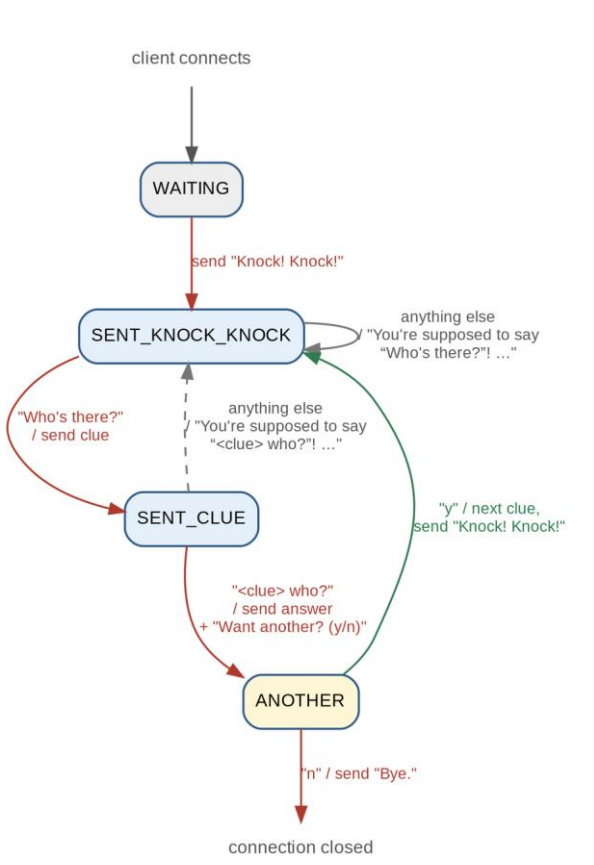


Figure 15.5 The knock-knock protocol as a state machine. Each arrow is labeled "what the client said / what the server answers"; grey arrows handle unexpected input; the green arrow starts the next joke.

Now the design decision that makes the rest of the chapter easy: **the protocol is an object, and it knows nothing about sockets**. **Listing 15.7** implements **Table 15.3** as a class whose one public method, `processInput`, takes the client's last line and returns the server's next line. The state is a private `enum` field; one object holds one conversation.

Listing 15.7 `KnockKnockProtocol.java`: the protocol of **Table 15.3** as a state machine, with a `main` that plays a scripted conversation.

```
/**
 * The knock-knock protocol as a state machine: one object per conversation.
 * It knows nothing about sockets, so it can be tested without a network.
 */
public class KnockKnockProtocol {
    private enum State { WAITING, SENT_KNOCK_KNOCK, SENT_CLUE, ANOTHER }

    private static final String[] CLUES = {"Lettuce", "Iron", "Dryer", "Atch"};
    private static final String[] ANSWERS = {
        "Lettuce in, it's cold out here!",
        "Iron your shirts? We already did!",
        "Dryer your tears, your laundry is ready!",
        "Bless you!"
    };

    private State state = State.WAITING;
    private int joke = 0;

    /** Returns the server's next line, given the client's last line (null at the start).
*/
    public String processInput(String input) {
        switch (state) {
            case WAITING:
                state = State.SENT_KNOCK_KNOCK;
                return "Knock! Knock!";
            case SENT_KNOCK_KNOCK:
                if (input.equalsIgnoreCase("Who's there?")) {
                    state = State.SENT_CLUE;
                    return CLUES[joke];
                }
                return "You're supposed to say \"Who's there?\"! Try again. Knock! Knock!";
            case SENT_CLUE:
                if (input.equalsIgnoreCase(CLUES[joke] + " who?")) {
                    state = State.ANOTHER;
                    return ANSWERS[joke] + " Want another? (y/n)";
                }
                state = State.SENT_KNOCK_KNOCK;
                return "You're supposed to say \"" + CLUES[joke]
                        + " who?\"! Try again. Knock! Knock!";
            case ANOTHER:
                if (input.equalsIgnoreCase("y")) {
                    joke = (joke + 1) % CLUES.length;
                    state = State.SENT_KNOCK_KNOCK;
                    return "Knock! Knock!";
                }
                state = State.WAITING;
                return "Bye.";
```

Listing 15.7 (Continued) *KnockKnockProtocol.java: the protocol of Table 15.3 as a state machine, with a main that plays a scripted conversation.*

```
            default:
                throw new IllegalStateException("unknown state " + state);
        }
    }

    public boolean isFinished(String serverLine) {
        return serverLine.equals("Bye.");
    }

    public static void main(String[] args) {
        // A scripted conversation, including two mistakes by the client.
        String[] clientLines = {
            "Who's there?", "Lettuce who?", "y",
            "Who is it?", "Who's there?", "Iron?", "Who's there?", "Iron who?", "n"
        };
        KnockKnockProtocol protocol = new KnockKnockProtocol();
        String serverLine = protocol.processInput(null);
        System.out.println("Server: " + serverLine);
        for (String clientLine : clientLines) {
            System.out.println("Client: " + clientLine);
            serverLine = protocol.processInput(clientLine);
            System.out.println("Server: " + serverLine);
            if (protocol.isFinished(serverLine)) {
                break;
            }
        }
    }
}
```

Its `main` method plays a whole conversation, including two mistakes, without any network:

```
Server: Knock! Knock!
Client: Who's there?
Server: Lettuce
Client: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
Client: y
Server: Knock! Knock!
Client: Who is it?
Server: You're supposed to say "Who's there?"! Try again. Knock! Knock!
Client: Who's there?
Server: Iron
Client: Iron?
Server: You're supposed to say "Iron who?"! Try again. Knock! Knock!
Client: Who's there?
Server: Iron
Client: Iron who?
Server: Iron your shirts? We already did! Want another? (y/n)
Client: n
Server: Bye.
```

Match the output against **Table 15.3**. After `y`, `joke` advances to the second clue. `Who is it?` is not `Who's there?`, so the server corrects the client and stays in `SENT_KNOCK_KNOCK`. `Iron?` is not `Iron who?`, so the protocol returns to `SENT_KNOCK_KNOCK` and starts that joke again, as the fifth row says. `n` ends with `Bye.`.

The `switch` mirrors the table row for row, one `case` per state and an unconditional `return` for "anything else": a program is easiest to check against its specification when it has the specification's shape. `isFinished` tells the socket code when to hang up without it knowing anything about jokes.

Listing 15.8 is the Python version, with `enum.Enum` and a chain of `if` statements.

Listing 15.8 `knock_knock_protocol.py`: the same state machine in Python.

```
"""knock_knock_protocol.py: the knock-knock protocol as a state machine.

One object per conversation. It knows nothing about sockets, so it can be
tested without a network.
"""
from enum import Enum, auto


class State(Enum):
    WAITING = auto()
    SENT_KNOCK_KNOCK = auto()
    SENT_CLUE = auto()
    ANOTHER = auto()


CLUES = ["Lettuce", "Iron", "Dryer", "Atch"]
ANSWERS = [
    "Lettuce in, it's cold out here!",
    "Iron your shirts? We already did!",
    "Dryer your tears, your laundry is ready!",
    "Bless you!",
]


class KnockKnockProtocol:
    def __init__(self):
        self._state = State.WAITING
        self._joke = 0

    def process_input(self, text):
        """Return the server's next line, given the client's last line (None at the
start)."""
        if self._state is State.WAITING:
            self._state = State.SENT_KNOCK_KNOCK
            return "Knock! Knock!"
        if self._state is State.SENT_KNOCK_KNOCK:
            if text.lower() == "who's there?":
                self._state = State.SENT_CLUE
                return CLUES[self._joke]
            return "You're supposed to say \"Who's there?\"! Try again. Knock! Knock!"
        if self._state is State.SENT_CLUE:
            if text.lower() == (CLUES[self._joke] + " who?").lower():
                self._state = State.ANOTHER
                return ANSWERS[self._joke] + " Want another? (y/n)"
            self._state = State.SENT_KNOCK_KNOCK
            return f"You're supposed to say \"{CLUES[self._joke]} who?\"! Try again. Knock!
Knock!"
        # State.ANOTHER
```

Listing 15.8 (Continued) *knock_knock_protocol.py: the same state machine in Python.*

```
        if text.lower() == "y":
            self._joke = (self._joke + 1) % len(CLUES)
            self._state = State.SENT_KNOCK_KNOCK
            return "Knock! Knock!"
        self._state = State.WAITING
        return "Bye."

    @staticmethod
    def is_finished(server_line):
        return server_line == "Bye."


def main():
    # A scripted conversation, including two mistakes by the client.
    client_lines = [
        "Who's there?", "Lettuce who?", "y",
        "Who is it?", "Who's there?", "Iron?", "Who's there?", "Iron who?", "n",
    ]
    protocol = KnockKnockProtocol()
    server_line = protocol.process_input(None)
    print("Server: " + server_line)
    for client_line in client_lines:
        print("Client: " + client_line)
        server_line = protocol.process_input(client_line)
        print("Server: " + server_line)
        if protocol.is_finished(server_line):
            break


if __name__ == "__main__":
    main()
```

It prints exactly what the Java version printed:

```
Server: Knock! Knock!
Client: Who's there?
Server: Lettuce
Client: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
Client: y
Server: Knock! Knock!
Client: Who is it?
Server: You're supposed to say "Who's there?"! Try again. Knock! Knock!
Client: Who's there?
Server: Iron
Client: Iron?
Server: You're supposed to say "Iron who?"! Try again. Knock! Knock!
Client: Who's there?
Server: Iron
Client: Iron who?
Server: Iron your shirts? We already did! Want another? (y/n)
Client: n
Server: Bye.
```

Why is the separation worth the extra class? It splits one hard problem into two easy ones. The protocol object can be tested with plain method calls, quickly and deterministically, with no ports, threads or timing. The socket code becomes a short loop that works for *any* line-based protocol: read a line, ask the protocol object, write its answer. In a team, the protocol's owner and the networking's owner can work in parallel, meeting only at `processInput`. The laundry service of Section 15.7 uses the same split, with the method `handle`.

> ✓ **KEY INSIGHT.** Write the protocol three times before you run it: as a table (**Table 15.3**), as a diagram (**Figure 15.5**), and as an object with no I/O (**Listing 15.7**). Each version catches mistakes the others hide, and the third one can be tested without a network.

15.6 Serving Many Clients at Once

The echo server of Section 15.4 serves one client and stops. A version that looped (accept a client, serve it until it leaves, accept the next) would be correct and nearly useless: while it talks to the first client nobody calls `accept`, so the second client waits in the backlog queue until the first says `bye`. If the first user goes to lunch, everyone waits for lunch.

The cure is the lesson of Chapter 14: do independent work on separate threads. The main loop only accepts; each accepted socket goes to another thread, and the loop returns to `accept` at once. **Figure 15.6** shows the structure.

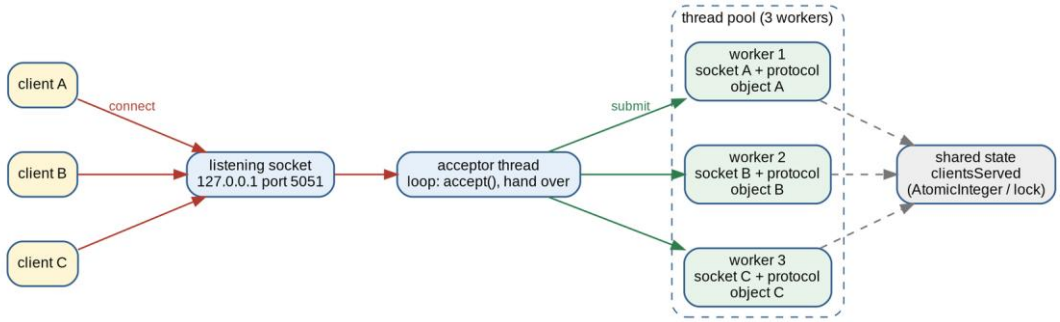


Figure 15.6 A multi-client server. The acceptor thread only accepts connections and hands each new socket to a worker of a fixed-size thread pool; each worker owns one socket and one protocol object; the only state the workers share is guarded by an atomic counter or a lock.

15.6.1 A thread per client, from a pool

The simplest way is `new Thread(...)` per connection, which the chat room of Exercise 15.11 uses. Its weakness is that every thread costs memory for its stack, working or waiting. With n clients and a stack of size s per thread, the stacks alone need about

$$M \approx n \cdot s \tag{15.3}$$

of memory: with the common $s = 1$ MB, a thousand idle clients reserve about 1 GB. Equation (15.3) is why servers use a **thread pool** (Chapter 14): a fixed number of workers taking jobs from a queue, so that under load new clients wait instead of exhausting memory. Java 21's *virtual threads* make one thread per client cheap again, and Python's `asyncio` serves many sockets from one thread; the design of **Figure 15.6** carries over to both [10].

15.6.2 The multi-client knock-knock server

Listing 15.9 is the knock-knock server with a thread pool, written as a class with `start` and `stop` so that it runs both from the command line and inside an automated test.

Listing 15.9 `KnockKnockServer.java`: a knock-knock server that talks to many clients at once, one pool thread per client, with a clean `stop()`.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.nio.charset.StandardCharsets;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

/** A knock-knock server that talks to many clients at once, one pool thread per client. */
public class KnockKnockServer {
    private static final int IDLE_TIMEOUT_MS = 60_000;

    private final ExecutorService pool;
    private final boolean verbose;
    private final AtomicInteger nextId = new AtomicInteger();
    private final AtomicInteger clientsServed = new AtomicInteger();
    private final Set<Socket> openSockets = ConcurrentHashMap.newKeySet();
    private ServerSocket serverSocket;
    private Thread acceptThread;
    private volatile boolean running;

    public KnockKnockServer(int maxClients, boolean verbose) {
        this.pool = Executors.newFixedThreadPool(maxClients);
        this.verbose = verbose;
    }

    /** Binds to the loopback address, starts accepting, and returns the actual port. */
    public int start(int port) throws IOException {
        serverSocket = new ServerSocket(port, 50, InetAddress.getLoopbackAddress());
        running = true;
        acceptThread = new Thread(this::acceptLoop, "acceptor");
        acceptThread.start();
        return serverSocket.getLocalPort();
    }
```

Listing 15.9 (Continued) *KnockKnockServer.java: a knock-knock server that talks to many clients at once, one pool thread per client, with a clean stop().*

```
    private void acceptLoop() {
        while (running) {
            try {
                Socket socket = serverSocket.accept();
                int id = nextId.incrementAndGet();
                log("client #" + id + " connected");
                pool.execute(() -> handle(socket, id));  // hand over, then accept the next
one
            } catch (SocketException e) {
                break;  // stop() closed the server socket
            } catch (IOException e) {
                log("accept failed: " + e.getMessage());
            }
        }
    }

    private void handle(Socket socket, int id) {
        openSockets.add(socket);
        try (socket;
             BufferedReader in = new BufferedReader(new InputStreamReader(
                     socket.getInputStream(), StandardCharsets.UTF_8));
             PrintWriter out = new PrintWriter(new OutputStreamWriter(
                     socket.getOutputStream(), StandardCharsets.UTF_8), true)) {
            socket.setSoTimeout(IDLE_TIMEOUT_MS);  // a silent client cannot hold a thread
forever
            KnockKnockProtocol protocol = new KnockKnockProtocol();  // one per
conversation
            out.println(protocol.processInput(null));
            String line;
            while ((line = in.readLine()) != null) {
                String reply = protocol.processInput(line);
                out.println(reply);
                if (protocol.isFinished(reply)) {
                    break;
                }
            }
        } catch (IOException e) {
            log("client #" + id + " dropped: " + e.getMessage());
        } finally {
            openSockets.remove(socket);
            clientsServed.incrementAndGet();
            log("client #" + id + " done");
        }
    }
```

Listing 15.9 (Continued) *KnockKnockServer.java: a knock-knock server that talks to many clients at once, one pool thread per client, with a clean stop().*

```
    /** Stops accepting, closes every open conversation, and waits for the pool. */
    public void stop() throws InterruptedException {
        running = false;
        try {
            serverSocket.close();  // makes accept() throw SocketException
        } catch (IOException ignored) {
            // closing is best effort
        }
        acceptThread.join();
        for (Socket socket : openSockets) {
            try {
                socket.close();
            } catch (IOException ignored) {
                // closing is best effort
            }
        }
        pool.shutdown();
        pool.awaitTermination(5, TimeUnit.SECONDS);
    }

    public int clientsServed() {
        return clientsServed.get();
    }

    private void log(String message) {
        if (verbose) {
            System.out.println(message);
        }
    }

    public static void main(String[] args) throws Exception {
        int port = args.length > 0 ? Integer.parseInt(args[0]) : 5051;
        KnockKnockServer server = new KnockKnockServer(8, true);
        int actual = server.start(port);
        System.out.println("KnockKnockServer listening on port " + actual + " (Ctrl+C stops
it)");
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            try {
                server.stop();
            } catch (InterruptedException ignored) {
                // we are exiting anyway
            }
            System.out.println("KnockKnockServer stopped after " + server.clientsServed()
                    + " clients");
        }));
    }
}
```

The class has four responsibilities, one method each.

- **`start(port)`** creates the listening socket on loopback, starts the `acceptor` thread, and returns the port actually bound (which matters for port 0). It returns at once, so the caller stays free.
- **`acceptLoop`** is the front door: it loops on `accept()` and hands each new socket to the pool with `pool.execute(() -> handle(socket, id))`. It never talks to a client itself, so a slow client cannot block the door.
- **`handle(socket, id)`** runs one whole conversation on a pool thread, with a *new* `KnockKnockProtocol`, so each client has its own state. It is the loop of Section 15.5: send the opening line, then read, ask the protocol, answer, until `Bye.`. `setSoTimeout(60_000)` stops a silent client from holding a worker forever, and `finally` counts the client however the conversation ends.

- **`stop()`** closes the listening socket (the blocked `accept()` throws `SocketException`, ending the loop), waits for the acceptor with `join`, closes every open socket to wake workers blocked in `readLine`, and shuts the pool down.

Fields touched by several threads are protected with Chapter 14's tools: `nextId` and `clientsServed` are `AtomicInteger`s, `openSockets` is a concurrent set, and `running` is `volatile`. Everything else, above all each client's protocol object, belongs to one thread and needs no protection. The principle: **share as little as possible, and protect what you must share.** `main` registers a *shutdown hook*, a thread the Java runtime runs on exit (for example after Ctrl+C), which stops the server and prints a summary.

Listing 15.10 is the Python server: `ThreadPoolExecutor` is the pool, a `threading.Lock` protects the counters and the socket set, and a `threading.Event` tells the accept loop to stop.

Listing 15.10 `knock_knock_server.py`: the multi-client server in Python.

```
"""knock_knock_server.py: a knock-knock server that talks to many clients at once."""
import socket
import sys
import threading
import time
from concurrent.futures import ThreadPoolExecutor

from knock_knock_protocol import KnockKnockProtocol

LOOPBACK = "127.0.0.1"
IDLE_TIMEOUT_S = 60


class KnockKnockServer:
    def __init__(self, max_clients=8, verbose=True):
        self._pool = ThreadPoolExecutor(max_workers=max_clients)
        self._verbose = verbose
        self._lock = threading.Lock()
        self._next_id = 0
        self._clients_served = 0
        self._open_sockets = set()
        self._stopping = threading.Event()
        self._server_socket = None
        self._accept_thread = None

    def start(self, port):
        """Bind to the loopback address, start accepting, and return the actual port."""
        self._server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        if sys.platform != "win32":  # reuse the port at once after a restart
            self._server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self._server_socket.bind((LOOPBACK, port))
        self._server_socket.listen(50)
        self._server_socket.settimeout(0.2)  # wake up regularly to check for stop()
        self._accept_thread = threading.Thread(target=self._accept_loop, name="acceptor")
        self._accept_thread.start()
        return self._server_socket.getsockname()[1]
```

Listing 15.10 (Continued) *knock_knock_server.py: the multi-client server in Python.*

```
    def _accept_loop(self):
        while not self._stopping.is_set():
            try:
                conn, _ = self._server_socket.accept()
            except socket.timeout:
                continue
            except OSError:
                break
            with self._lock:
                self._next_id += 1
                client_id = self._next_id
            self._log(f"client #{client_id} connected")
            self._pool.submit(self._handle, conn, client_id)  # hand over, then accept the
next one

    def _handle(self, conn, client_id):
        with self._lock:
            self._open_sockets.add(conn)
        try:
            conn.settimeout(IDLE_TIMEOUT_S)  # a silent client cannot hold a thread forever
            with conn, conn.makefile("r", encoding="utf-8", newline="\n") as reader, \
                    conn.makefile("w", encoding="utf-8", newline="\n") as writer:
                protocol = KnockKnockProtocol()  # one per conversation
                writer.write(protocol.process_input(None) + "\n")
                writer.flush()
                for line in reader:
                    reply = protocol.process_input(line.rstrip("\r\n"))
                    writer.write(reply + "\n")
                    writer.flush()
                    if protocol.is_finished(reply):
                        break
        except OSError as e:
            self._log(f"client #{client_id} dropped: {e}")
        finally:
            with self._lock:
                self._open_sockets.discard(conn)
                self._clients_served += 1
            self._log(f"client #{client_id} done")
```

Listing 15.10 (Continued) *knock_knock_server.py: the multi-client server in Python.*

```
    def stop(self):
        """Stop accepting, close every open conversation, and wait for the pool."""
        self._stopping.set()
        self._accept_thread.join()
        self._server_socket.close()
        with self._lock:
            sockets = list(self._open_sockets)
        for conn in sockets:
            try:
                conn.shutdown(socket.SHUT_RDWR)  # wakes a thread blocked in readline()
            except OSError:
                pass
        self._pool.shutdown(wait=True)

    def clients_served(self):
        with self._lock:
            return self._clients_served

    def _log(self, message):
        if self._verbose:
            print(message, flush=True)


def main():
    port = int(sys.argv[1]) if len(sys.argv) > 1 else 5051
    server = KnockKnockServer(max_clients=8, verbose=True)
    actual = server.start(port)
    print(f"KnockKnockServer listening on port {actual} (Ctrl+C stops it)", flush=True)
    try:
        while True:
            time.sleep(1)  # sleep until Ctrl+C
    except KeyboardInterrupt:
        pass
    server.stop()
    print(f"KnockKnockServer stopped after {server.clients_served()} clients")


if __name__ == "__main__":
    main()
```

One difference is forced by the operating systems: closing a listening socket from another thread does not reliably wake a Python thread blocked in `accept()`. So the listening socket gets a timeout of 0.2 seconds, and the loop checks the stop event after each timeout. `stop()` wakes blocked workers with `shutdown(SHUT_RDWR)`, which makes `readline` return `""`. The standard library's `socketserver.ThreadingTCPServer` does much of this for you; we write it out once to show every moving part.

15.6.3 The client, and three terminals

Listing 15.11 is the client for humans. The server speaks first, so the loop begins by reading.

Listing 15.11 `KnockKnockClient.java`: talk to the knock-knock server from the keyboard.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ConnectException;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class KnockKnockClient {
    public static void main(String[] args) {
        String host = args.length > 0 ? args[0] : "localhost";
        int port = args.length > 1 ? Integer.parseInt(args[1]) : 5051;

        try (Socket socket = new Socket(host, port);
             PrintWriter out = new PrintWriter(new OutputStreamWriter(
                     socket.getOutputStream(), StandardCharsets.UTF_8), true);
             BufferedReader in = new BufferedReader(new InputStreamReader(
                     socket.getInputStream(), StandardCharsets.UTF_8));
             BufferedReader keyboard = new BufferedReader(new InputStreamReader(
                     System.in, StandardCharsets.UTF_8))) {
            String fromServer;
            while ((fromServer = in.readLine()) != null) {  // the server speaks first
                System.out.println("Server: " + fromServer);
                if (fromServer.equals("Bye.")) {
                    break;
                }
                String fromUser = keyboard.readLine();
                if (fromUser == null) {
                    break;  // end of input: just hang up
                }
                System.out.println("Client: " + fromUser);
                out.println(fromUser);
            }
        } catch (ConnectException e) {
            System.out.println("Cannot connect to " + host + " port " + port
                    + ": is the server running?");
        } catch (IOException e) {
            System.out.println("Connection lost: " + e.getMessage());
        }
    }
}
```

Listing 15.12 is the Python client.

Listing 15.12 `knock_knock_client.py`: the knock-knock client in Python.

```python
"""knock_knock_client.py: talk to the knock-knock server from the keyboard."""
import socket
import sys


def main():
    host = sys.argv[1] if len(sys.argv) > 1 else "localhost"
    port = int(sys.argv[2]) if len(sys.argv) > 2 else 5051

    try:
        with socket.create_connection((host, port)) as sock, \
             sock.makefile("w", encoding="utf-8", newline="\n") as writer, \
             sock.makefile("r", encoding="utf-8", newline="\n") as reader:
            for from_server in reader:  # the server speaks first
                from_server = from_server.rstrip("\r\n")
                print("Server: " + from_server)
                if from_server == "Bye.":
                    break
                from_user = sys.stdin.readline()
                if from_user == "":
                    break  # end of input: just hang up
                from_user = from_user.rstrip("\r\n")
                print("Client: " + from_user)
                writer.write(from_user + "\n")
                writer.flush()
    except ConnectionRefusedError:
        print(f"Cannot connect to {host} port {port}: is the server running?")
    except OSError as e:
        print(f"Connection lost: {e}")


if __name__ == "__main__":
    main()
```

Start the server with `java KnockKnockServer 5051` and run as many clients as you like from other terminals. For the book we ran the Java client and then the Python client against the Java server, each fed `KnockKnockClient.input.txt` (in Python, `knock_knock_client.input.txt`), which contains a deliberate mistake:

```
Who's there?
Dryer who?
Who's there?
Lettuce who?
n
```

Each client printed:

```
Server: Knock! Knock!
Client: Who's there?
Server: Lettuce
Client: Dryer who?
Server: You're supposed to say "Lettuce who?"! Try again. Knock! Knock!
Client: Who's there?
Server: Lettuce
Client: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
Client: n
Server: Bye.
```

The server's terminal showed both clients, and after Ctrl+C its shutdown hook printed the summary:

```
KnockKnockServer listening on port 5051 (Ctrl+C stops it)
client #1 connected
client #1 done
client #2 connected
client #2 done
KnockKnockServer stopped after 2 clients
```

The Python server, `python3 knock_knock_server.py 5051`, printed the same log for the same two clients:

```
KnockKnockServer listening on port 5051 (Ctrl+C stops it)
client #1 connected
client #1 done
client #2 connected
client #2 done
KnockKnockServer stopped after 2 clients
```

15.6.4 Proving that the server is concurrent

Three terminals show that clients *can* connect, not that they are served *at the same time*; a sequential server would produce the same transcripts. **Listing 15.13** tests the claim properly. It starts the server in the same program on an ephemeral port, runs three scripted clients on three threads, and stops the server. Each client, once greeted, waits at a `CyclicBarrier` (Chapter 14) until *all three* have been greeted. A sequential server would never greet the second client while the first waited, the barrier would time out after five seconds, and the test would print `false`.

Listing 15.13 `KnockKnockLoadTest.java`: start a server, run three clients at the same time against it, and stop it.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

/** Starts a server, runs three clients at the same time against it, and stops it. */
public class KnockKnockLoadTest {
    public static void main(String[] args) throws Exception {
        KnockKnockServer server = new KnockKnockServer(3, false);
        int port = server.start(0);  // 0: an ephemeral port chosen by the OS
        System.out.println("server started on an ephemeral port");

        List<List<String>> scripts = List.of(
                List.of("Who's there?", "Lettuce who?", "n"),
                List.of("Who's there?", "Lettuce who?", "y", "Who's there?", "Iron who?",
"n"),
                List.of("Hello?", "Who's there?", "Lettuce who?", "n"));
        // Every client waits here until all of them have been greeted.
        CyclicBarrier together = new CyclicBarrier(scripts.size());
        ExecutorService clients = Executors.newFixedThreadPool(scripts.size());
        List<Future<List<String>>> results = new ArrayList<>();
        for (List<String> script : scripts) {
            results.add(clients.submit(() -> converse(port, script, together)));
        }

        boolean allGreetedTogether = true;
        for (int i = 0; i < results.size(); i++) {
            System.out.println("--- client " + (char) ('A' + i) + " ---");
            try {
                results.get(i).get().forEach(System.out::println);
            } catch (ExecutionException e) {
                allGreetedTogether = false;
                System.out.println("failed: " + e.getCause().getClass().getSimpleName());
            }
        }
        clients.shutdown();
```

Listing 15.13 (Continued) *KnockKnockLoadTest.java: start a server, run three clients at the same time against it, and stop it.*

```
        server.stop();
        System.out.println("server stopped; clients served: " + server.clientsServed());
        System.out.println("all clients were greeted before any finished: " +
allGreetedTogether);
    }

    static List<String> converse(int port, List<String> script, CyclicBarrier together)
            throws Exception {
        List<String> transcript = new ArrayList<>();
        try (Socket socket = new Socket(InetAddress.getLoopbackAddress(), port);
             BufferedReader in = new BufferedReader(new InputStreamReader(
                     socket.getInputStream(), StandardCharsets.UTF_8));
             PrintWriter out = new PrintWriter(new OutputStreamWriter(
                     socket.getOutputStream(), StandardCharsets.UTF_8), true)) {
            socket.setSoTimeout(5000);
            transcript.add("Server: " + in.readLine());
            together.await(5, TimeUnit.SECONDS);
            for (String mine : script) {
                transcript.add("Client: " + mine);
                out.println(mine);
                transcript.add("Server: " + in.readLine());
            }
        }
        return transcript;
    }
}
```

```
server started on an ephemeral port
--- client A ---
Server: Knock! Knock!
Client: Who's there?
Server: Lettuce
Client: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
Client: n
Server: Bye.
--- client B ---
Server: Knock! Knock!
Client: Who's there?
Server: Lettuce
Client: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
Client: y
Server: Knock! Knock!
Client: Who's there?
Server: Iron
Client: Iron who?
Server: Iron your shirts? We already did! Want another? (y/n)
Client: n
Server: Bye.
--- client C ---
Server: Knock! Knock!
Client: Hello?
Server: You're supposed to say "Who's there?"! Try again. Knock! Knock!
Client: Who's there?
Server: Lettuce
Client: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
Client: n
Server: Bye.
server stopped; clients served: 3
all clients were greeted before any finished: true
```

The transcripts are collected with `Future.get()` and printed only at the end, so the output order is fixed although the conversations overlapped. Client C is corrected independently of A and B, confirming that each connection has its own protocol object. **Listing 15.14** is the Python test, with `threading.Barrier`:

Listing 15.14 `knock_knock_load_test.py`: the same concurrency test in Python.

```
"""knock_knock_load_test.py: start a server, run three clients at once, stop it."""
import socket
import threading
from concurrent.futures import ThreadPoolExecutor

from knock_knock_server import KnockKnockServer


def converse(port, script, together):
    transcript = []
    with socket.create_connection(("127.0.0.1", port), timeout=5) as sock, \
         sock.makefile("r", encoding="utf-8", newline="\n") as reader, \
         sock.makefile("w", encoding="utf-8", newline="\n") as writer:
        transcript.append("Server: " + reader.readline().rstrip("\n"))
        together.wait(timeout=5)
        for mine in script:
            transcript.append("Client: " + mine)
            writer.write(mine + "\n")
            writer.flush()
            transcript.append("Server: " + reader.readline().rstrip("\n"))
    return transcript


def main():
    server = KnockKnockServer(max_clients=3, verbose=False)
    port = server.start(0)  # 0: an ephemeral port chosen by the OS
    print("server started on an ephemeral port")

    scripts = [
        ["Who's there?", "Lettuce who?", "n"],
        ["Who's there?", "Lettuce who?", "y", "Who's there?", "Iron who?", "n"],
        ["Hello?", "Who's there?", "Lettuce who?", "n"],
    ]
    # Every client waits here until all of them have been greeted.
    together = threading.Barrier(len(scripts))
    with ThreadPoolExecutor(max_workers=len(scripts)) as clients:
        futures = [clients.submit(converse, port, script, together) for script in scripts]
```

Listing 15.14 (Continued) *knock_knock_load_test.py: the same concurrency test in Python.*

```
        all_greeted_together = True
        for i, future in enumerate(futures):
            print(f"--- client {chr(ord('A') + i)} ---")
            try:
                for line in future.result():
                    print(line)
            except Exception as e:  # the barrier was broken or timed out
                all_greeted_together = False
                print(f"failed: {type(e).__name__}")
    server.stop()
    print(f"server stopped; clients served: {server.clients_served()}")
    print(f"all clients were greeted before any finished:
{str(all_greeted_together).lower()}")


if __name__ == "__main__":
    main()
```

```
server started on an ephemeral port
--- client A ---
Server: Knock! Knock!
Client: Who's there?
Server: Lettuce
Client: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
Client: n
Server: Bye.
--- client B ---
Server: Knock! Knock!
Client: Who's there?
Server: Lettuce
Client: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
Client: y
Server: Knock! Knock!
Client: Who's there?
Server: Iron
Client: Iron who?
Server: Iron your shirts? We already did! Want another? (y/n)
Client: n
Server: Bye.
--- client C ---
Server: Knock! Knock!
Client: Hello?
Server: You're supposed to say "Who's there?"! Try again. Knock! Knock!
Client: Who's there?
Server: Lettuce
Client: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
Client: n
Server: Bye.
server stopped; clients served: 3
all clients were greeted before any finished: true
```

Exercise 15.5 asks you to shrink the pool to two workers and predict what happens; the answer is a good test of whether **Figure 15.6** has sunk in.

15.7 Profitina Laundry: an Order-Status Service in JSON over TCP

Time to build something the laundry can use. Customers keep phoning the counter to ask whether their laundry is ready. The owner wants a server in the back room that knows every order's status and a client on the counter laptop that asks it, and staff should be able to move an order forward from their own client. Server and clients may be written by different people in different languages, so the protocol must be precise and language-neutral.

15.7.1 The specification, on paper

We keep one line per message and a protocol testable without a network, and make one change: each line carries one **JSON** object.

> **DEFINITION. JSON** (JavaScript Object Notation) is a text format for structured data [11]. A JSON value is an *object* `{"key": value, ...}`, an *array* `[value, ...]`, a *string* in double quotes, a *number*, `true`, `false` or `null`. Objects and arrays nest. Keys are always strings.

JSON is understood by every modern language and readable by humans, and because a message is a set of named fields, a field added later does not break old clients. One JSON object per line, with no line breaks inside, is the convention called *JSON Lines*; it combines JSON's structure with newline framing.

Table 15.4 is the complete specification. The client always speaks first, every request gets exactly one response line, and every response has `ok`; a refusal carries a readable `error`.

Table 15.4 The order-status protocol. One JSON object per line in each direction; the client speaks first; every request receives exactly one response.

Request (client → server)	Successful response (server → client)	Refusals
`{"cmd":"ping"}`	`{"ok":true,"reply":"pong","server":"java"}` (or `"python"`)	—
`{"cmd":"status","order":"ORD-0713","phone_last4":"7890"}`	`{"ok":true,"order":…,"customer":"B*** S******","status":"WASHING","ready_by":…}`	same error for an unknown order and a wrong phone
`{"cmd":"advance","order":…,"token":…}`	`{"ok":true,"order":…,"status":"DRYING"}`	wrong or missing staff token; order already picked up; no such order
`{"cmd":"bye"}`	`{"ok":true,"reply":"bye"}`, then the server closes	—
`{"cmd":"shutdown","token":…}`	`{"ok":true,"reply":"shutting down"}`, then the server stops	wrong or missing staff token
a line that is not a JSON object	—	`{"ok":false,"error":"malformed request"}`
a line longer than 1024 characters	—	`{"ok":false,"error":"request too long"}`, then close

Statuses follow Chapter 1's life cycle, `RECEIVED`, `WASHING`, `DRYING`, `IRONING`, `READY`, `PICKED_UP`, and `advance` moves an order one step. **Figure 15.7** shows a typical conversation.

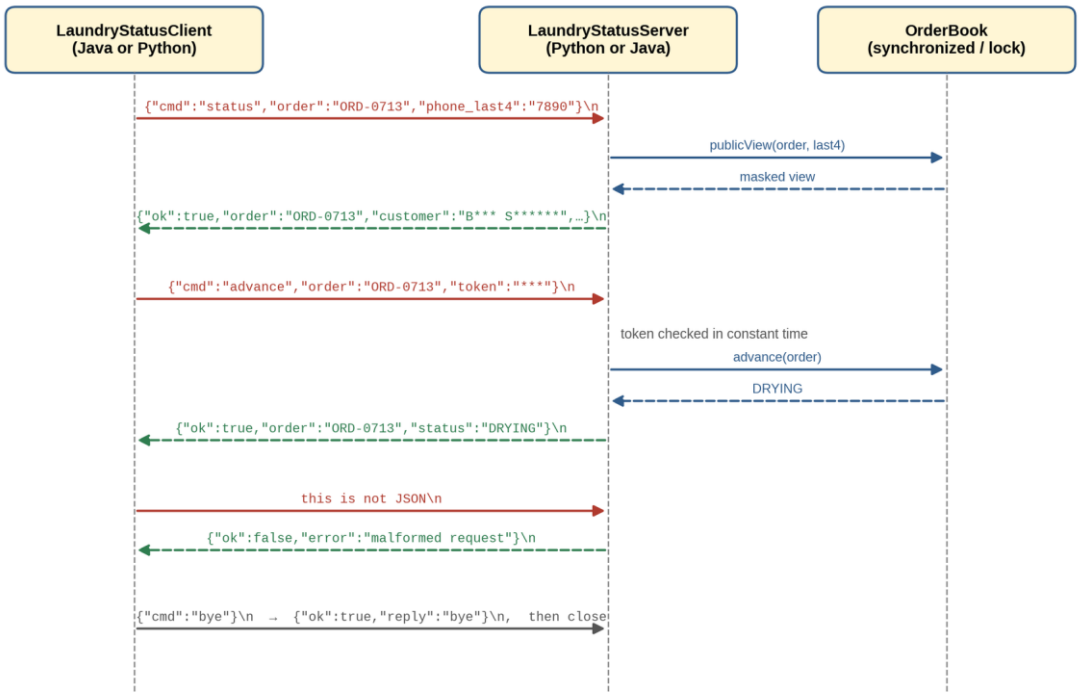


Figure 15.7 A conversation with the order-status service. Requests travel as JSON lines; the server consults the shared `OrderBook`, which is guarded by a lock; a malformed line receives an error, not silence; `bye` ends the conversation. The client may be Java and the server Python, or the other way round.

Four choices protect people rather than programs (Section 15.9 explains them): a customer must give the last four phone digits, so a lost receipt reveals nothing; the response shows only initials, never the phone number; a wrong order number and a wrong phone number get the *same* error, so a guesser cannot learn which orders exist; and staff commands need a secret token kept in an environment variable, never in the code.

15.7.2 JSON in both languages

Python has JSON in its standard library (`json.loads`, `json.dumps`). Java's standard library, surprisingly, has none; real projects add Jackson or Gson. To avoid downloads, and to show there is no magic, **Listing 15.15** is a small, complete JSON reader and writer of under 300 lines: a textbook *recursive-descent parser*, one method per kind of value, each calling `value()` again for nested values.

Listing 15.15 `Json.java`: a minimal JSON reader and writer; objects become `LinkedHashMap`, arrays `ArrayList`, whole numbers `Long`.

```
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/**
 * A minimal JSON reader and writer, enough for a line-based protocol.
 * Objects become LinkedHashMap (keys keep their order), arrays ArrayList,
 * whole numbers Long, other numbers Double.
 */
public final class Json {
    private final String text;
    private int pos = 0;

    private Json(String text) {
        this.text = text;
    }

    /** Parses one JSON value; throws IllegalArgumentException if the text is not JSON. */
    public static Object parse(String text) {
        Json parser = new Json(text);
        parser.skipSpaces();
        Object value = parser.value();
        parser.skipSpaces();
        if (parser.pos != text.length()) {
            throw parser.error("extra text after the value");
        }
        return value;
    }

    /** Builds an object from alternating keys and values: object("ok", true, "n", 3). */
    public static Map<String, Object> object(Object... keysAndValues) {
        Map<String, Object> map = new LinkedHashMap<>();
        for (int i = 0; i < keysAndValues.length; i += 2) {
            map.put((String) keysAndValues[i], keysAndValues[i + 1]);
        }
        return map;
    }

    /** Writes a value compactly, with no spaces, as Python's json.dumps(separators=(",",
":")). */
    public static String write(Object value) {
        StringBuilder sb = new StringBuilder();
        writeValue(sb, value);
        return sb.toString();
    }
```

Listing 15.15 (Continued) *Json.java: a minimal JSON reader and writer; objects become LinkedHashMap, arrays ArrayList, whole numbers Long.*

```
    // ---------- reading ----------

    private Object value() {
        if (pos >= text.length()) {
            throw error("unexpected end of text");
        }
        char c = text.charAt(pos);
        switch (c) {
            case '{': return object();
            case '[': return array();
            case '"': return string();
            case 't': return literal("true", Boolean.TRUE);
            case 'f': return literal("false", Boolean.FALSE);
            case 'n': return literal("null", null);
            default:
                if (c == '-' || Character.isDigit(c)) {
                    return number();
                }
                throw error("unexpected character '" + c + "'");
        }
    }

    private Map<String, Object> object() {
        Map<String, Object> map = new LinkedHashMap<>();
        pos++;  // skip {
        skipSpaces();
        if (peek() == '}') {
            pos++;
            return map;
        }
        while (true) {
            skipSpaces();
            if (peek() != '"') {
                throw error("expected a key in double quotes");
            }
            String key = string();
            skipSpaces();
            expect(':');
            skipSpaces();
            map.put(key, value());
            skipSpaces();
            if (peek() == ',') {
                pos++;
            } else {
                expect('}');
                return map;
```

Listing 15.15 (Continued) *Json.java: a minimal JSON reader and writer; objects become LinkedHashMap, arrays ArrayList, whole numbers Long.*

```
            }
        }
    }

    private List<Object> array() {
        List<Object> list = new ArrayList<>();
        pos++;  // skip [
        skipSpaces();
        if (peek() == ']') {
            pos++;
            return list;
        }
        while (true) {
            skipSpaces();
            list.add(value());
            skipSpaces();
            if (peek() == ',') {
                pos++;
            } else {
                expect(']');
                return list;
            }
        }
    }

    private String string() {
        StringBuilder sb = new StringBuilder();
        pos++;  // skip the opening quote
        while (true) {
            if (pos >= text.length()) {
                throw error("unterminated string");
            }
            char c = text.charAt(pos++);
            if (c == '"') {
                return sb.toString();
            } else if (c == '\\') {
                if (pos >= text.length()) {
                    throw error("unterminated escape");
                }
                char e = text.charAt(pos++);
                switch (e) {
                    case '"': sb.append('"'); break;
                    case '\\': sb.append('\\'); break;
                    case '/': sb.append('/'); break;
                    case 'b': sb.append('\b'); break;
                    case 'f': sb.append('\f'); break;
```

Listing 15.15 (Continued) *Json.java: a minimal JSON reader and writer; objects become LinkedHashMap, arrays ArrayList, whole numbers Long.*

```
                    case 'n': sb.append('\n'); break;
                    case 'r': sb.append('\r'); break;
                    case 't': sb.append('\t'); break;
                    case 'u':
                        if (pos + 4 > text.length()) {
                            throw error("short \\u escape");
                        }
                        sb.append((char) Integer.parseInt(text.substring(pos, pos + 4),
16));
                        pos += 4;
                        break;
                    default: throw error("bad escape \\" + e);
                }
            } else if (c < 0x20) {
                throw error("control character inside a string");
            } else {
                sb.append(c);
            }
        }
    }

    private Object number() {
        int start = pos;
        if (peek() == '-') {
            pos++;
        }
        while (pos < text.length() && "0123456789.eE+-".indexOf(text.charAt(pos)) >= 0) {
            pos++;
        }
        String s = text.substring(start, pos);
        try {
            if (s.contains(".") || s.contains("e") || s.contains("E")) {
                return Double.parseDouble(s);
            }
            return Long.parseLong(s);
        } catch (NumberFormatException e) {
            throw error("bad number " + s);
        }
    }
```

Listing 15.15 (Continued) *Json.java: a minimal JSON reader and writer; objects become LinkedHashMap, arrays ArrayList, whole numbers Long.*

```
    private Object literal(String word, Object result) {
        if (!text.startsWith(word, pos)) {
            throw error("expected " + word);
        }
        pos += word.length();
        return result;
    }

    private char peek() {
        return pos < text.length() ? text.charAt(pos) : '\0';
    }

    private void expect(char c) {
        if (peek() != c) {
            throw error("expected '" + c + "'");
        }
        pos++;
    }

    private void skipSpaces() {
        while (pos < text.length() && " \t\r\n".indexOf(text.charAt(pos)) >= 0) {
            pos++;
        }
    }

    private IllegalArgumentException error(String message) {
        return new IllegalArgumentException("malformed JSON at position " + pos + ": " +
message);
    }

    // ---------- writing ----------

    private static void writeValue(StringBuilder sb, Object value) {
        if (value == null) {
            sb.append("null");
        } else if (value instanceof String s) {
            writeString(sb, s);
        } else if (value instanceof Boolean || value instanceof Long || value instanceof
Integer
                || value instanceof Double) {
            sb.append(value);
        } else if (value instanceof Map<?, ?> map) {
            sb.append('{');
            boolean first = true;
            for (Map.Entry<?, ?> entry : map.entrySet()) {
                if (!first) {
                    sb.append(',');
                }
```

Listing 15.15 (Continued) *Json.java: a minimal JSON reader and writer; objects become LinkedHashMap, arrays ArrayList, whole numbers Long.*

```
                first = false;
                writeString(sb, (String) entry.getKey());
                sb.append(':');
                writeValue(sb, entry.getValue());
            }
            sb.append('}');
        } else if (value instanceof List<?> list) {
            sb.append('[');
            for (int i = 0; i < list.size(); i++) {
                if (i > 0) {
                    sb.append(',');
                }
                writeValue(sb, list.get(i));
            }
            sb.append(']');
        } else {
            throw new IllegalArgumentException("cannot write " +
value.getClass().getSimpleName());
        }
    }

    private static void writeString(StringBuilder sb, String s) {
        sb.append('"');
        for (char c : s.toCharArray()) {
            switch (c) {
                case '"': sb.append("\\\""); break;
                case '\\': sb.append("\\\\"); break;
                case '\n': sb.append("\\n"); break;
                case '\r': sb.append("\\r"); break;
                case '\t': sb.append("\\t"); break;
                case '\b': sb.append("\\b"); break;
                case '\f': sb.append("\\f"); break;
                default:
                    if (c < 0x20) {
                        sb.append(String.format("\\u%04x", (int) c));
                    } else {
                        sb.append(c);
                    }
            }
        }
        sb.append('"');
    }
```

Listing 15.15 (Continued) *Json.java: a minimal JSON reader and writer; objects become LinkedHashMap, arrays ArrayList, whole numbers Long.*

```
    public static void main(String[] args) {
        String[] samples = {
            "{\"cmd\":\"status\",\"order\":\"ORD-0713\",\"phone_last4\":\"7890\"}",
            "{ \"ok\" : true, \"kg\" : 3.5, \"price\" : 24500, \"items\" : [\"shirt\",
\"towel\"] }",
            "{\"note\":\"line 1\\nline 2 \\\"quoted\\\" caf\\u00e9\", \"paid\":null}",
            "{\"cmd\":\"status\",\"order\":}",
            "{\"cmd\":\"ping\"} extra"
        };
        for (String sample : samples) {
            System.out.println("text   : " + sample);
            try {
                Object value = parse(sample);
                System.out.println("written: " + write(value));
            } catch (IllegalArgumentException e) {
                System.out.println("rejected: not valid JSON");
            }
        }
    }
}
```

```
text   : {"cmd":"status","order":"ORD-0713","phone_last4":"7890"}
written: {"cmd":"status","order":"ORD-0713","phone_last4":"7890"}
text   : { "ok" : true, "kg" : 3.5, "price" : 24500, "items" : ["shirt", "towel"] }
written: {"ok":true,"kg":3.5,"price":24500,"items":["shirt","towel"]}
text   : {"note":"line 1\nline 2 \"quoted\" caf\u00e9", "paid":null}
written: {"note":"line 1\nline 2 \"quoted\" café","paid":null}
text   : {"cmd":"status","order":}
rejected: not valid JSON
text   : {"cmd":"ping"} extra
rejected: not valid JSON
```

The class offers three static methods. `parse(text)` returns a `Map`, `List`, `String`, `Long`, `Double`, `Boolean` or `null`, and throws `IllegalArgumentException` for anything that is not exactly one JSON value (the fourth sample lacks a value, the fifth has extra text). `write(value)` produces compact text with no spaces. `object(...)` builds messages: `Json.object("cmd", "ping")`. Maps are `LinkedHashMap`s, so fields are written in insertion order. The reader decodes escapes such as `\n`, `\"` and `\u00e9` (`é`), and the writer produces them, which guarantees that a message never contains a raw newline.

Listing 15.16 performs the same round trips with Python's `json` module.

Listing 15.16 `json_roundtrip.py`: the same round trips with Python's standard `json` module.

```
"""json_roundtrip.py: read and write the protocol's JSON with the standard json module."""
import json


def write(value):
    """Write compactly, with no spaces, exactly as Json.write in Java does."""
    return json.dumps(value, separators=(",", ":"), ensure_ascii=False)


def main():
    samples = [
        '{"cmd":"status","order":"ORD-0713","phone_last4":"7890"}',
        '{ "ok" : true, "kg" : 3.5, "price" : 24500, "items" : ["shirt", "towel"] }',
        '{"note":"line 1\\nline 2 \\"quoted\\" caf\\u00e9", "paid":null}',
        '{"cmd":"status","order":}',
        '{"cmd":"ping"} extra',
    ]
    for sample in samples:
        print("text   : " + sample)
        try:
            value = json.loads(sample)
            print("written: " + write(value))
        except json.JSONDecodeError:
            print("rejected: not valid JSON")


if __name__ == "__main__":
    main()
```

```
text   : {"cmd":"status","order":"ORD-0713","phone_last4":"7890"}
written: {"cmd":"status","order":"ORD-0713","phone_last4":"7890"}
text   : { "ok" : true, "kg" : 3.5, "price" : 24500, "items" : ["shirt", "towel"] }
written: {"ok":true,"kg":3.5,"price":24500,"items":["shirt","towel"]}
text   : {"note":"line 1\nline 2 \"quoted\" caf\u00e9", "paid":null}
written: {"note":"line 1\nline 2 \"quoted\" café","paid":null}
text   : {"cmd":"status","order":}
rejected: not valid JSON
text   : {"cmd":"ping"} extra
rejected: not valid JSON
```

The two outputs are identical, and that is a requirement, not a coincidence: `separators=(",", ":")` makes Python write no spaces, like `Json.write`, and `ensure_ascii=False` writes `é` as itself. Since both sides write the same text for the same value, either can read the other's messages. (If `é` shows as `?`, your terminal is not in UTF-8; on Windows, `chcp 65001` fixes it.)

15.7.3 The server

Listing 15.17 is the server: the knock-knock skeleton (listening socket, acceptor, pool, clean `stop`) plus three domain classes in the same file, `Status`, `LaundryOrder` and `OrderBook`.

Listing 15.17 `LaundryStatusServer.java`: the Profitina Laundry order-status service, with its domain classes `Status`, `LaundryOrder` and `OrderBook`.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.Reader;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

/** The Profitina Laundry order-status service: one JSON object per line, both ways. */
public class LaundryStatusServer {
    static final int MAX_LINE = 1024;
    private static final int IDLE_TIMEOUT_MS = 60_000;

    private final OrderBook book;
    private final String staffToken;  // null: staff commands are disabled
    private final boolean verbose;
    private final ExecutorService pool = Executors.newFixedThreadPool(16);
    private final AtomicInteger nextId = new AtomicInteger();
    private final Set<Socket> openSockets = ConcurrentHashMap.newKeySet();
    private final CountDownLatch stopped = new CountDownLatch(1);
    private ServerSocket serverSocket;
    private volatile boolean running;

    LaundryStatusServer(OrderBook book, String staffToken, boolean verbose) {
        this.book = book;
        this.staffToken = staffToken;
        this.verbose = verbose;
    }
```

Listing 15.17 (Continued) *LaundryStatusServer.java: the Profitina Laundry order-status service, with its domain classes Status, LaundryOrder and OrderBook.*

```
    /** A server holding the three sample orders used throughout this chapter. */
    public static LaundryStatusServer withSampleOrders(String staffToken, boolean verbose)
{
        return new LaundryStatusServer(OrderBook.sample(), staffToken, verbose);
    }

    public int start(int port) throws IOException {
        serverSocket = new ServerSocket(port, 50, InetAddress.getLoopbackAddress());
        running = true;
        new Thread(this::acceptLoop, "acceptor").start();
        return serverSocket.getLocalPort();
    }

    private void acceptLoop() {
        while (running) {
            try {
                Socket socket = serverSocket.accept();
                int id = nextId.incrementAndGet();
                pool.execute(() -> serve(socket, id));
            } catch (SocketException e) {
                break;  // stop() closed the server socket
            } catch (IOException e) {
                log("accept failed: " + e.getMessage());
            }
        }
    }

    private void serve(Socket socket, int id) {
        openSockets.add(socket);
        log("client #" + id + " connected");
        try (socket;
             Reader in = new BufferedReader(new InputStreamReader(
                     socket.getInputStream(), StandardCharsets.UTF_8));
             PrintWriter out = new PrintWriter(new OutputStreamWriter(
                     socket.getOutputStream(), StandardCharsets.UTF_8), true)) {
            socket.setSoTimeout(IDLE_TIMEOUT_MS);
            String line;
            while ((line = readLimitedLine(in)) != null) {
                if (line.length() > MAX_LINE) {
                    out.println(Json.write(error("request too long")));
                    break;  // do not read the rest of a flood
                }
                Map<String, Object> request = parseRequest(line);
                String cmd = request == null ? "?" : String.valueOf(request.get("cmd"));
                Map<String, Object> response = handle(request);
                out.println(Json.write(response));
                log("client #" + id + ": " + cmd + " -> " +
(response.get("ok").equals(true)
```

Listing 15.17 (Continued) *LaundryStatusServer.java: the Profitina Laundry order-status service, with its domain classes Status, LaundryOrder and OrderBook.*

```
                        ? "ok" : "refused"));  // never log personal data
                if (cmd.equals("bye")) {
                    break;
                }
                if (cmd.equals("shutdown") && response.get("ok").equals(true)) {
                    new Thread(this::stop).start();  // not from a pool thread: stop()
waits for the pool
                    break;
                }
            }
        } catch (IOException e) {
            log("client #" + id + " dropped: " + e.getMessage());
        } finally {
            openSockets.remove(socket);
            log("client #" + id + " disconnected");
        }
    }

    /** Reads one line, but stops reading after MAX_LINE + 1 characters. */
    static String readLimitedLine(Reader in) throws IOException {
        StringBuilder sb = new StringBuilder();
        int c;
        while ((c = in.read()) != -1) {
            if (c == '\n') {
                return sb.toString();
            }
            if (c != '\r') {
                sb.append((char) c);
            }
            if (sb.length() > MAX_LINE) {
                return sb.toString();  // too long: the caller refuses it
            }
        }
        return sb.length() > 0 ? sb.toString() : null;
    }

    @SuppressWarnings("unchecked")
    private static Map<String, Object> parseRequest(String line) {
        try {
            Object value = Json.parse(line);
            return value instanceof Map ? (Map<String, Object>) value : null;
        } catch (IllegalArgumentException e) {
            return null;
        }
    }
```

Listing 15.17 (Continued) *LaundryStatusServer.java: the Profitina Laundry order-status service, with its domain classes Status, LaundryOrder and OrderBook.*

```
    /** The whole protocol, as a function from request to response. */
    Map<String, Object> handle(Map<String, Object> request) {
        if (request == null) {
            return error("malformed request");
        }
        Object cmd = request.get("cmd");
        if ("ping".equals(cmd)) {
            return Json.object("ok", true, "reply", "pong", "server", "java");
        } else if ("status".equals(cmd)) {
            Object order = request.get("order");
            Object last4 = request.get("phone_last4");
            if (!(order instanceof String) || !(last4 instanceof String)) {
                return error("status needs order and phone_last4");
            }
            Map<String, Object> view = book.publicView((String) order, (String) last4);
            // The same answer for "no such order" and "wrong phone": no hints to a
guesser.
            return view != null ? view : error("no order with that number and phone");
        } else if ("advance".equals(cmd)) {
            if (!isStaff(request.get("token"))) {
                return error("not authorized");
            }
            Object order = request.get("order");
            if (!(order instanceof String)) {
                return error("advance needs order");
            }
            try {
                Status status = book.advance((String) order);
                return Json.object("ok", true, "order", order, "status", status.name());
            } catch (IllegalArgumentException | IllegalStateException e) {
                return error(e.getMessage());
            }
        } else if ("bye".equals(cmd)) {
            return Json.object("ok", true, "reply", "bye");
        } else if ("shutdown".equals(cmd)) {
            return isStaff(request.get("token"))
                    ? Json.object("ok", true, "reply", "shutting down")
                    : error("not authorized");
        }
        return error("unknown command: " + cmd);
    }
```

Listing 15.17 (Continued) *LaundryStatusServer.java: the Profitina Laundry order-status service, with its domain classes Status, LaundryOrder and OrderBook.*

```
    private boolean isStaff(Object token) {
        if (staffToken == null || !(token instanceof String)) {
            return false;
        }
        // Constant-time comparison: the time taken does not reveal how much matched.
        return MessageDigest.isEqual(staffToken.getBytes(StandardCharsets.UTF_8),
                ((String) token).getBytes(StandardCharsets.UTF_8));
    }

    private static Map<String, Object> error(String message) {
        return Json.object("ok", false, "error", message);
    }

    public void stop() {
        if (!running) {
            return;
        }
        running = false;
        try {
            serverSocket.close();
        } catch (IOException ignored) {
            // closing is best effort
        }
        for (Socket socket : openSockets) {
            try {
                socket.close();
            } catch (IOException ignored) {
                // closing is best effort
            }
        }
        pool.shutdown();
        try {
            pool.awaitTermination(5, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        stopped.countDown();
    }

    public void awaitStop() throws InterruptedException {
        stopped.await();
    }
```

Listing 15.17 (Continued) *LaundryStatusServer.java: the Profitina Laundry order-status service, with its domain classes Status, LaundryOrder and OrderBook.*

```
    private void log(String message) {
        if (verbose) {
            System.out.println(message);
        }
    }

    public static void main(String[] args) throws Exception {
        int port = args.length > 0 ? Integer.parseInt(args[0]) : 5052;
        String token = System.getenv("PROFITINA_STAFF_TOKEN");  // a secret never lives in
code
        boolean verbose = !(args.length > 1 && args[1].equals("--quiet"));
        LaundryStatusServer server = withSampleOrders(token, verbose);
        int actual = server.start(port);
        System.out.println("LaundryStatusServer (java) ready on port " + actual);
        if (token == null) {
            System.out.println("staff commands disabled: PROFITINA_STAFF_TOKEN is not
set");
        }
        Runtime.getRuntime().addShutdownHook(new Thread(server::stop));  // Ctrl+C
        server.awaitStop();
        System.out.println("LaundryStatusServer (java) stopped");
    }
}

enum Status {
    RECEIVED, WASHING, DRYING, IRONING, READY, PICKED_UP;

    Status next() {
        if (this == PICKED_UP) {
            throw new IllegalStateException("order already picked up");
        }
        return values()[ordinal() + 1];
    }
}

class LaundryOrder {
    final String id;
    final String customer;
    final String phone;
    final String readyBy;
    Status status;
```

Listing 15.17 (Continued) *LaundryStatusServer.java: the Profitina Laundry order-status service, with its domain classes Status, LaundryOrder and OrderBook.*

```
    LaundryOrder(String id, String customer, String phone, String readyBy, Status status) {
        this.id = id;
        this.customer = customer;
        this.phone = phone;
        this.readyBy = readyBy;
        this.status = status;
    }
}

/** All orders; every method is synchronized because many client threads share one book. */
class OrderBook {
    private final Map<String, LaundryOrder> orders = new HashMap<>();

    synchronized void add(LaundryOrder order) {
        orders.put(order.id, order);
    }

    /** What a customer may see: no phone number, and only the initials of the name. */
    synchronized Map<String, Object> publicView(String id, String phoneLast4) {
        LaundryOrder o = orders.get(id);
        if (o == null || !o.phone.endsWith(phoneLast4) || phoneLast4.length() != 4) {
            return null;
        }
        return Json.object("ok", true, "order", o.id, "customer", mask(o.customer),
                "status", o.status.name(), "ready_by", o.readyBy);
    }

    synchronized Status advance(String id) {
        LaundryOrder o = orders.get(id);
        if (o == null) {
            throw new IllegalArgumentException("no such order");
        }
        o.status = o.status.next();
        return o.status;
    }
```

Listing 15.17 (Continued) *LaundryStatusServer.java: the Profitina Laundry order-status service, with its domain classes Status, LaundryOrder and OrderBook.*

```
    static String mask(String name) {
        StringBuilder sb = new StringBuilder();
        for (String word : name.split(" ")) {
            if (sb.length() > 0) {
                sb.append(' ');
            }
            sb.append(word.charAt(0)).append("*".repeat(word.length() - 1));
        }
        return sb.toString();
    }

    static OrderBook sample() {
        OrderBook book = new OrderBook();
        book.add(new LaundryOrder("ORD-0713", "Budi Santoso", "081234567890",
                "2026-09-24 16:00", Status.WASHING));
        book.add(new LaundryOrder("ORD-0714", "Sari Wulandari", "081298765432",
                "2026-09-23 18:00", Status.READY));
        book.add(new LaundryOrder("ORD-0715", "Andi Pratama", "085711112222",
                "2026-09-25 10:00", Status.RECEIVED));
        return book;
    }
}
```

Started with the staff token in the environment (Section 15.7.4), visited by one client running the scenario of **Listing 15.19**, and stopped with Ctrl+C, it printed:

```
LaundryStatusServer (java) ready on port 5052
client #1 connected
client #1: ping -> ok
client #1: status -> ok
client #1: status -> refused
client #1: status -> refused
client #1: ? -> refused
client #1: advance -> refused
client #1: advance -> ok
client #1: status -> ok
client #1: bye -> ok
client #1 disconnected
LaundryStatusServer (java) stopped
```

- **`serve(socket, id)`** is the per-client loop. `readLimitedLine` gives up after 1025 characters, because `readLine` has no limit and a hostile client could send a gigabyte without a newline; an over-long line is refused and the connection closed. A line that is not a JSON object becomes `null`, which `handle` answers with `malformed request`, so the client always gets an answer. The log records the command and its outcome, never order numbers, names or phone digits.
- **`handle(request)`** is the whole of **Table 15.4** as one function from request map to response map, with no I/O, so it can be tested without a network. Every field is checked for existence and type (`instanceof String`) before use: `{"cmd":"status","order":42}` must be refused, not crash the worker with a `ClassCastException`.
- **`isStaff(token)`** compares with `MessageDigest.isEqual`, which takes the same time wherever the difference lies; `equals` stops at the first difference, and timing many attempts could reveal a secret character by character. The token comes from the environment variable `PROFITINA_STAFF_TOKEN`; if it is unset, every staff command is refused.
- **`shutdown`** must not call `stop()` on the worker thread that received it: `stop()` waits for the pool, and that worker *is* in the pool, so it would wait for itself forever. It starts a fresh thread to call `stop()` instead.
- **`OrderBook`** holds the orders in a `HashMap`, which is not safe for concurrent use, so every method is `synchronized` (Chapter 14). `publicView` builds the customer's view, with `mask` turning "Budi Santoso" into "B*** S******"; `advance` uses `Status.next()`, which throws at `PICKED_UP`. The package-private helpers are reached from other files through the factory `withSampleOrders`.

`main` prints the "ready" line, registers a shutdown hook for Ctrl+C, and waits on a `CountDownLatch` that `stop()` releases, so the program ends after a `shutdown` command or Ctrl+C. Without the token, it says so at startup:

```
LaundryStatusServer (java) ready on port 5052
staff commands disabled: PROFITINA_STAFF_TOKEN is not set
LaundryStatusServer (java) stopped
```

Listing 15.18 is the Python server, class for class; `dataclass` writes `LaundryOrder`'s constructor, a `threading.Lock` replaces `synchronized`, and `hmac.compare_digest` is the constant-time comparison.

Listing 15.18 `laundry_status_server.py`: the order-status service in Python.

```
"""laundry_status_server.py: the Profitina Laundry order-status service.

One JSON object per line, both ways. Run: python3 laundry_status_server.py [port] [--quiet]
"""
import hmac
import json
import os
import socket
import sys
import threading
from concurrent.futures import ThreadPoolExecutor
from dataclasses import dataclass
from enum import Enum

LOOPBACK = "127.0.0.1"
MAX_LINE = 1024
IDLE_TIMEOUT_S = 60


class Status(Enum):
    RECEIVED = 0
    WASHING = 1
    DRYING = 2
    IRONING = 3
    READY = 4
    PICKED_UP = 5

    def next(self):
        if self is Status.PICKED_UP:
            raise ValueError("order already picked up")
        return Status(self.value + 1)


@dataclass
class LaundryOrder:
    id: str
    customer: str
    phone: str
    ready_by: str
    status: Status


def mask(name):
    return " ".join(word[0] + "*" * (len(word) - 1) for word in name.split(" "))
```

Listing 15.18 (Continued) *laundry_status_server.py: the order-status service in Python.*

```
class OrderBook:
    """All orders; every method takes the lock because many client threads share one
book."""

    def __init__(self):
        self._orders = {}
        self._lock = threading.Lock()

    def add(self, order):
        with self._lock:
            self._orders[order.id] = order

    def public_view(self, order_id, phone_last4):
        """What a customer may see: no phone number, and only the initials of the name."""
        with self._lock:
            o = self._orders.get(order_id)
            if o is None or len(phone_last4) != 4 or not o.phone.endswith(phone_last4):
                return None
            return {"ok": True, "order": o.id, "customer": mask(o.customer),
                    "status": o.status.name, "ready_by": o.ready_by}

    def advance(self, order_id):
        with self._lock:
            o = self._orders.get(order_id)
            if o is None:
                raise ValueError("no such order")
            o.status = o.status.next()
            return o.status

    @staticmethod
    def sample():
        book = OrderBook()
        book.add(LaundryOrder("ORD-0713", "Budi Santoso", "081234567890",
                              "2026-09-24 16:00", Status.WASHING))
        book.add(LaundryOrder("ORD-0714", "Sari Wulandari", "081298765432",
                              "2026-09-23 18:00", Status.READY))
        book.add(LaundryOrder("ORD-0715", "Andi Pratama", "085711112222",
                              "2026-09-25 10:00", Status.RECEIVED))
        return book


def write(value):
    return json.dumps(value, separators=(",", ":"), ensure_ascii=False)
```

Listing 15.18 (Continued) *laundry_status_server.py: the order-status service in Python.*

```
def error(message):
    return {"ok": False, "error": message}


class LaundryStatusServer:
    def __init__(self, book, staff_token, verbose=True):
        self._book = book
        self._staff_token = staff_token  # None: staff commands are disabled
        self._verbose = verbose
        self._pool = ThreadPoolExecutor(max_workers=16)
        self._lock = threading.Lock()
        self._next_id = 0
        self._open_sockets = set()
        self._stopping = threading.Event()
        self._stopped = threading.Event()
        self._server_socket = None
        self._accept_thread = None

    def start(self, port):
        self._server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        if sys.platform != "win32":  # reuse the port at once after a restart
            self._server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self._server_socket.bind((LOOPBACK, port))
        self._server_socket.listen(50)
        self._server_socket.settimeout(0.2)  # wake up regularly to check for stop()
        self._accept_thread = threading.Thread(target=self._accept_loop, name="acceptor")
        self._accept_thread.start()
        return self._server_socket.getsockname()[1]

    def _accept_loop(self):
        while not self._stopping.is_set():
            try:
                conn, _ = self._server_socket.accept()
            except socket.timeout:
                continue
            except OSError:
                break
            with self._lock:
                self._next_id += 1
                client_id = self._next_id
            self._pool.submit(self._serve, conn, client_id)
```

Listing 15.18 (Continued) *laundry_status_server.py: the order-status service in Python.*

```
    def _serve(self, conn, client_id):
        with self._lock:
            self._open_sockets.add(conn)
        self._log(f"client #{client_id} connected")
        try:
            conn.settimeout(IDLE_TIMEOUT_S)
            with conn, conn.makefile("r", encoding="utf-8", newline="\n") as reader, \
                    conn.makefile("w", encoding="utf-8", newline="\n") as writer:
                while True:
                    line = reader.readline(MAX_LINE + 1)  # never read more than this
                    if line == "":
                        break  # the client closed the connection
                    line = line.rstrip("\r\n")
                    if len(line) > MAX_LINE:
                        writer.write(write(error("request too long")) + "\n")
                        writer.flush()
                        break  # do not read the rest of a flood
                    request = parse_request(line)
                    cmd = "?" if request is None else str(request.get("cmd"))
                    response = self.handle(request)
                    writer.write(write(response) + "\n")
                    writer.flush()
                    outcome = "ok" if response["ok"] else "refused"
                    self._log(f"client #{client_id}: {cmd} -> {outcome}")  # never log
personal data
                    if cmd == "bye":
                        break
                    if cmd == "shutdown" and response["ok"]:
                        # not from a pool thread: stop() waits for the pool
                        threading.Thread(target=self.stop).start()
                        break
        except OSError as e:
            self._log(f"client #{client_id} dropped: {e}")
        finally:
            with self._lock:
                self._open_sockets.discard(conn)
            self._log(f"client #{client_id} disconnected")
```

Listing 15.18 (Continued) *laundry_status_server.py: the order-status service in Python.*

```
    def handle(self, request):
        """The whole protocol, as a function from request to response."""
        if request is None:
            return error("malformed request")
        cmd = request.get("cmd")
        if cmd == "ping":
            return {"ok": True, "reply": "pong", "server": "python"}
        if cmd == "status":
            order, last4 = request.get("order"), request.get("phone_last4")
            if not isinstance(order, str) or not isinstance(last4, str):
                return error("status needs order and phone_last4")
            view = self._book.public_view(order, last4)
            # The same answer for "no such order" and "wrong phone": no hints to a guesser.
            return view if view is not None else error("no order with that number and
phone")
        if cmd == "advance":
            if not self._is_staff(request.get("token")):
                return error("not authorized")
            order = request.get("order")
            if not isinstance(order, str):
                return error("advance needs order")
            try:
                status = self._book.advance(order)
            except ValueError as e:
                return error(str(e))
            return {"ok": True, "order": order, "status": status.name}
        if cmd == "bye":
            return {"ok": True, "reply": "bye"}
        if cmd == "shutdown":
            if self._is_staff(request.get("token")):
                return {"ok": True, "reply": "shutting down"}
            return error("not authorized")
        return error(f"unknown command: {'null' if cmd is None else cmd}")

    def _is_staff(self, token):
        if self._staff_token is None or not isinstance(token, str):
            return False
        # Constant-time comparison: the time taken does not reveal how much matched.
        return hmac.compare_digest(self._staff_token.encode("utf-8"), token.encode("utf-
8"))
```

Listing 15.18 (Continued) *laundry_status_server.py: the order-status service in Python.*

```
    def stop(self):
        if self._stopping.is_set():
            return
        self._stopping.set()
        self._accept_thread.join()
        self._server_socket.close()
        with self._lock:
            sockets = list(self._open_sockets)
        for conn in sockets:
            try:
                conn.shutdown(socket.SHUT_RDWR)
            except OSError:
                pass
        self._pool.shutdown(wait=True)
        self._stopped.set()

    def wait_until_stopped(self):
        while not self._stopped.wait(0.5):  # short waits keep Ctrl+C responsive
            pass

    def _log(self, message):
        if self._verbose:
            print(message, flush=True)


def parse_request(line):
    try:
        value = json.loads(line)
    except json.JSONDecodeError:
        return None
    return value if isinstance(value, dict) else None


def main():
    args = sys.argv[1:]
    port = int(args[0]) if args else 5052
    verbose = "--quiet" not in args
    token = os.environ.get("PROFITINA_STAFF_TOKEN")  # a secret never lives in code
    server = LaundryStatusServer(OrderBook.sample(), token, verbose)
    actual = server.start(port)
    print(f"LaundryStatusServer (python) ready on port {actual}", flush=True)
    if token is None:
        print("staff commands disabled: PROFITINA_STAFF_TOKEN is not set", flush=True)
    try:
        server.wait_until_stopped()
    except KeyboardInterrupt:  # Ctrl+C
```

Listing 15.18 (Continued) *laundry_status_server.py: the order-status service in Python.*

```
        server.stop()
    print("LaundryStatusServer (python) stopped")


if __name__ == "__main__":
    main()
```

```
LaundryStatusServer (python) ready on port 5052
client #1 connected
client #1: ping -> ok
client #1: status -> ok
client #1: status -> refused
client #1: status -> refused
client #1: ? -> refused
client #1: advance -> refused
client #1: advance -> ok
client #1: status -> ok
client #1: bye -> ok
client #1 disconnected
LaundryStatusServer (python) stopped
```

Python's `readline(MAX_LINE + 1)` takes a limit directly. The log is the same; the only visible difference between the servers is the `"server"` field of `ping`, added so that a client can tell which implementation answered.

15.7.4 The client object

A client could build JSON strings by hand, but that would scatter the protocol over every program that uses the service. The object-oriented answer is a class that *represents the service* inside the client: `client.status("ORD-0713", "7890")` looks like an ordinary call, and the class turns it into a request line and the response line into a map. This is the **Proxy** pattern, a local object standing in for a remote one [12]. **Listing 15.19** is the proxy in Java, with a `main` that runs a fixed scenario.

Listing 15.19 `LaundryStatusClient.java`: a client object that hides the wire format, and a scenario that exercises every rule of **Table 15.4**.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ConnectException;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.util.LinkedHashMap;
import java.util.Map;

/** A client object that hides the wire format: callers see methods, not JSON lines. */
public class LaundryStatusClient implements AutoCloseable {
    private final Socket socket;
    private final BufferedReader in;
    private final PrintWriter out;

    public LaundryStatusClient(String host, int port) throws IOException {
        socket = new Socket();
        socket.connect(new InetSocketAddress(host, port), 2000);  // give up after 2 s
        socket.setSoTimeout(5000);  // and never wait more than 5 s for an answer
        in = new BufferedReader(new InputStreamReader(
                socket.getInputStream(), StandardCharsets.UTF_8));
        out = new PrintWriter(new OutputStreamWriter(
                socket.getOutputStream(), StandardCharsets.UTF_8), true);
    }

    public Map<String, Object> ping() throws IOException {
        return send(Json.object("cmd", "ping"));
    }

    public Map<String, Object> status(String order, String phoneLast4) throws IOException {
        return send(Json.object("cmd", "status", "order", order, "phone_last4",
phoneLast4));
    }

    public Map<String, Object> advance(String order, String token) throws IOException {
        return send(Json.object("cmd", "advance", "order", order, "token", token));
    }

    public Map<String, Object> shutdownServer(String token) throws IOException {
        return send(Json.object("cmd", "shutdown", "token", token));
    }
```

Listing 15.19 (Continued) *LaundryStatusClient.java: a client object that hides the wire format, and a scenario that exercises every rule of Table 15.4.*

```
    public Map<String, Object> send(Map<String, Object> request) throws IOException {
        return sendRaw(Json.write(request));
    }

    /** Sends one line exactly as given; useful for testing how the server copes with junk.
*/
    @SuppressWarnings("unchecked")
    public Map<String, Object> sendRaw(String line) throws IOException {
        out.println(line);
        String reply = in.readLine();
        if (reply == null) {
            throw new IOException("the server closed the connection");
        }
        return (Map<String, Object>) Json.parse(reply);
    }

    @Override
    public void close() throws IOException {
        try {
            if (!socket.isClosed()) {
                send(Json.object("cmd", "bye"));
            }
        } catch (IOException ignored) {
            // the server may already be gone; we are leaving anyway
        } finally {
            socket.close();
        }
    }

    /** Shows a message as JSON, with any token replaced by ***: secrets never reach a log.
*/
    static String show(Map<String, Object> message) {
        Map<String, Object> copy = new LinkedHashMap<>(message);
        if (copy.containsKey("token")) {
            copy.put("token", "***");
        }
        return Json.write(copy);
    }

    private static void exchange(LaundryStatusClient client, Map<String, Object> request)
            throws IOException {
        System.out.println("> " + show(request));
        System.out.println("< " + show(client.send(request)));
    }
```

Listing 15.19 (Continued) *LaundryStatusClient.java: a client object that hides the wire format, and a scenario that exercises every rule of Table 15.4.*

```
    /** The same conversation, whichever server is at the other end. */
    static void runScenario(LaundryStatusClient client, String token) throws IOException {
        exchange(client, Json.object("cmd", "ping"));
        exchange(client, Json.object("cmd", "status", "order", "ORD-0713", "phone_last4",
"7890"));
        exchange(client, Json.object("cmd", "status", "order", "ORD-0713", "phone_last4",
"0000"));
        exchange(client, Json.object("cmd", "status", "order", "ORD-9999", "phone_last4",
"7890"));
        System.out.println("> this is not JSON");
        System.out.println("< " + show(client.sendRaw("this is not JSON")));
        exchange(client, Json.object("cmd", "advance", "order", "ORD-0713", "token",
"guess"));
        exchange(client, Json.object("cmd", "advance", "order", "ORD-0713", "token",
                token == null ? "" : token));
        exchange(client, Json.object("cmd", "status", "order", "ORD-0713", "phone_last4",
"7890"));
    }

    public static void main(String[] args) {
        String host = args.length > 0 ? args[0] : "localhost";
        int port = args.length > 1 ? Integer.parseInt(args[1]) : 5052;
        String token = System.getenv("PROFITINA_STAFF_TOKEN");
        try (LaundryStatusClient client = new LaundryStatusClient(host, port)) {
            runScenario(client, token);
        } catch (ConnectException e) {
            System.out.println("Cannot connect to " + host + " port " + port
                    + ": is the server running?");
        } catch (IOException e) {
            System.out.println("Connection problem: " + e.getMessage());
        }
    }
}
```

The constructor connects with a two-second limit (`new Socket(host, port)` could wait long for an unreachable machine) and limits every read to five seconds. `send` writes one JSON line and reads one, which is the whole protocol from the client's side; `sendRaw` lets tests send deliberate junk; `close()` says `bye` and ignores failures, since a leaving client should not crash because the server left first. `show` prints messages with any `token` replaced by `***`: secrets never belong in logs, screens or screenshots.

To run the pair, set the staff token in both terminals first. It is a secret: choose your own long random string, and never commit it:

```
Linux, macOS:        export PROFITINA_STAFF_TOKEN=your-own-long-random-string
Windows PowerShell:  $env:PROFITINA_STAFF_TOKEN = "your-own-long-random-string"
Windows cmd:         set PROFITINA_STAFF_TOKEN=your-own-long-random-string

Terminal 1:  java LaundryStatusServer 5052
Terminal 2:  java LaundryStatusClient localhost 5052
```

The client printed each request after `>` and each response after `<`:

```
> {"cmd":"ping"}
< {"ok":true,"reply":"pong","server":"java"}
> {"cmd":"status","order":"ORD-0713","phone_last4":"7890"}
< {"ok":true,"order":"ORD-0713","customer":"B***
S******","status":"WASHING","ready_by":"2026-09-24 16:00"}
> {"cmd":"status","order":"ORD-0713","phone_last4":"0000"}
< {"ok":false,"error":"no order with that number and phone"}
> {"cmd":"status","order":"ORD-9999","phone_last4":"7890"}
< {"ok":false,"error":"no order with that number and phone"}
> this is not JSON
< {"ok":false,"error":"malformed request"}
> {"cmd":"advance","order":"ORD-0713","token":"***"}
< {"ok":false,"error":"not authorized"}
> {"cmd":"advance","order":"ORD-0713","token":"***"}
< {"ok":true,"order":"ORD-0713","status":"DRYING"}
> {"cmd":"status","order":"ORD-0713","phone_last4":"7890"}
< {"ok":true,"order":"ORD-0713","customer":"B***
S******","status":"DRYING","ready_by":"2026-09-24 16:00"}
```

Read the scenario against **Table 15.4**: the right phone digits show the masked name; the wrong digits and the unknown order get exactly the same refusal; a non-JSON line gets `malformed request` and the connection stays usable; the guessed token is refused and the real one moves the order from `WASHING` to `DRYING`.

Listing 15.20 is the Python proxy; `__enter__` and `__exit__` make it work in a `with` statement.

Listing 15.20 `laundry_status_client.py`: the client object in Python.

```
"""laundry_status_client.py: a client object that hides the wire format."""
import json
import os
import socket
import sys


def write(value):
    return json.dumps(value, separators=(",", ":"), ensure_ascii=False)


class LaundryStatusClient:
    def __init__(self, host, port):
        self._sock = socket.create_connection((host, port), timeout=2)  # give up after 2 s
        self._sock.settimeout(5)  # and never wait more than 5 s for an answer
        self._reader = self._sock.makefile("r", encoding="utf-8", newline="\n")
        self._writer = self._sock.makefile("w", encoding="utf-8", newline="\n")

    def ping(self):
        return self.send({"cmd": "ping"})

    def status(self, order, phone_last4):
        return self.send({"cmd": "status", "order": order, "phone_last4": phone_last4})

    def advance(self, order, token):
        return self.send({"cmd": "advance", "order": order, "token": token})

    def shutdown_server(self, token):
        return self.send({"cmd": "shutdown", "token": token})

    def send(self, request):
        return self.send_raw(write(request))

    def send_raw(self, line):
        """Send one line exactly as given; useful for testing how the server copes with
junk."""
        self._writer.write(line + "\n")
        self._writer.flush()
        reply = self._reader.readline()
        if reply == "":
            raise ConnectionError("the server closed the connection")
        return json.loads(reply)
```

Listing 15.20 (Continued) *laundry_status_client.py: the client object in Python.*

```
    def close(self):
        try:
            self.send({"cmd": "bye"})
        except OSError:
            pass  # the server may already be gone; we are leaving anyway
        finally:
            self._reader.close()
            self._writer.close()
            self._sock.close()

    def __enter__(self):
        return self

    def __exit__(self, *exc):
        self.close()


def show(message):
    """Show a message as JSON, with any token replaced by ***: secrets never reach a
log."""
    copy = dict(message)
    if "token" in copy:
        copy["token"] = "***"
    return write(copy)


def _exchange(client, request):
    print("> " + show(request))
    print("< " + show(client.send(request)))


def run_scenario(client, token):
    """The same conversation, whichever server is at the other end."""
    _exchange(client, {"cmd": "ping"})
    _exchange(client, {"cmd": "status", "order": "ORD-0713", "phone_last4": "7890"})
    _exchange(client, {"cmd": "status", "order": "ORD-0713", "phone_last4": "0000"})
    _exchange(client, {"cmd": "status", "order": "ORD-9999", "phone_last4": "7890"})
    print("> this is not JSON")
    print("< " + show(client.send_raw("this is not JSON")))
    _exchange(client, {"cmd": "advance", "order": "ORD-0713", "token": "guess"})
    _exchange(client, {"cmd": "advance", "order": "ORD-0713", "token": token or ""})
    _exchange(client, {"cmd": "status", "order": "ORD-0713", "phone_last4": "7890"})
```

Listing 15.20 (Continued) *laundry_status_client.py: the client object in Python.*

```
def main():
    host = sys.argv[1] if len(sys.argv) > 1 else "localhost"
    port = int(sys.argv[2]) if len(sys.argv) > 2 else 5052
    token = os.environ.get("PROFITINA_STAFF_TOKEN")
    try:
        with LaundryStatusClient(host, port) as client:
            run_scenario(client, token)
    except ConnectionRefusedError:
        print(f"Cannot connect to {host} port {port}: is the server running?")
    except OSError as e:
        print(f"Connection problem: {e}")


if __name__ == "__main__":
    main()
```

Against the Python server (`python3 laundry_status_server.py 5052`), it printed:

```
> {"cmd":"ping"}
< {"ok":true,"reply":"pong","server":"python"}
> {"cmd":"status","order":"ORD-0713","phone_last4":"7890"}
< {"ok":true,"order":"ORD-0713","customer":"B***
S******","status":"WASHING","ready_by":"2026-09-24 16:00"}
> {"cmd":"status","order":"ORD-0713","phone_last4":"0000"}
< {"ok":false,"error":"no order with that number and phone"}
> {"cmd":"status","order":"ORD-9999","phone_last4":"7890"}
< {"ok":false,"error":"no order with that number and phone"}
> this is not JSON
< {"ok":false,"error":"malformed request"}
> {"cmd":"advance","order":"ORD-0713","token":"***"}
< {"ok":false,"error":"not authorized"}
> {"cmd":"advance","order":"ORD-0713","token":"***"}
< {"ok":true,"order":"ORD-0713","status":"DRYING"}
> {"cmd":"status","order":"ORD-0713","phone_last4":"7890"}
< {"ok":true,"order":"ORD-0713","customer":"B***
S******","status":"DRYING","ready_by":"2026-09-24 16:00"}
```

15.7.5 Java talks to Python, and Python talks to Java

We have claimed that the protocol, not the language, decides who can talk to whom. **Listing 15.21** demonstrates it: a Java program starts the **Python** server as a separate process on an ephemeral port, runs the Java client's scenario against it, stops it with `shutdown`, and checks that it exited.

Listing 15.21 `InteropDemo.java`: start the Python server as a separate process and talk to it with the Java client.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.security.SecureRandom;
import java.util.HexFormat;
import java.util.concurrent.TimeUnit;

/** Starts the PYTHON server as a separate process and talks to it with the JAVA client. */
public class InteropDemo {
    public static void main(String[] args) throws Exception {
        String script = args.length > 0 ? args[0] : "../python/laundry_status_server.py";
        boolean windows =
System.getProperty("os.name").toLowerCase().startsWith("windows");
        String python = System.getenv().getOrDefault("PYTHON", windows ? "python" :
"python3");

        // A fresh random token for this run only; it is handed over in the environment.
        byte[] random = new byte[16];
        new SecureRandom().nextBytes(random);
        String token = HexFormat.of().formatHex(random);

        ProcessBuilder builder = new ProcessBuilder(python, script, "0", "--quiet");
        builder.environment().put("PROFITINA_STAFF_TOKEN", token);
        builder.redirectErrorStream(true);
        Process server = builder.start();

        BufferedReader serverOut = new BufferedReader(new InputStreamReader(
                server.getInputStream(), StandardCharsets.UTF_8));
        int port = -1;
        String line;
        while ((line = serverOut.readLine()) != null) {
            if (line.startsWith("LaundryStatusServer")) {  // "... ready on port 43127"
                port = Integer.parseInt(line.substring(line.lastIndexOf(' ') + 1));
                break;
            }
        }
        if (port < 0) {
            System.out.println("the Python server did not start");
            return;
        }
        Thread drain = new Thread(() -> serverOut.lines().forEach(s -> { }));  // keep the
pipe empty
        drain.setDaemon(true);
        drain.start();
        System.out.println("started the Python server as a separate process");
```

Listing 15.21 (Continued) *InteropDemo.java: start the Python server as a separate process and talk to it with the Java client.*

```
        try (LaundryStatusClient client = new LaundryStatusClient("127.0.0.1", port)) {
            System.out.println("answered by: " + client.ping().get("server"));
            LaundryStatusClient.runScenario(client, token);
            System.out.println("> " + LaundryStatusClient.show(Json.object("cmd",
"shutdown",
                    "token", token)));
            System.out.println("< " +
LaundryStatusClient.show(client.shutdownServer(token)));
        }
        boolean exited = server.waitFor(10, TimeUnit.SECONDS);
        System.out.println("Python server exited: " + exited + ", exit code "
                + (exited ? server.exitValue() : -1));
    }
}
```

Run it from the `java` folder with `java InteropDemo` (it expects the Python files in `../python` and runs `python3`, or `python` on Windows, unless `PYTHON` names another interpreter):

```
started the Python server as a separate process
answered by: python
> {"cmd":"ping"}
< {"ok":true,"reply":"pong","server":"python"}
> {"cmd":"status","order":"ORD-0713","phone_last4":"7890"}
< {"ok":true,"order":"ORD-0713","customer":"B***
S******","status":"WASHING","ready_by":"2026-09-24 16:00"}
> {"cmd":"status","order":"ORD-0713","phone_last4":"0000"}
< {"ok":false,"error":"no order with that number and phone"}
> {"cmd":"status","order":"ORD-9999","phone_last4":"7890"}
< {"ok":false,"error":"no order with that number and phone"}
> this is not JSON
< {"ok":false,"error":"malformed request"}
> {"cmd":"advance","order":"ORD-0713","token":"***"}
< {"ok":false,"error":"not authorized"}
> {"cmd":"advance","order":"ORD-0713","token":"***"}
< {"ok":true,"order":"ORD-0713","status":"DRYING"}
> {"cmd":"status","order":"ORD-0713","phone_last4":"7890"}
< {"ok":true,"order":"ORD-0713","customer":"B***
S******","status":"DRYING","ready_by":"2026-09-24 16:00"}
> {"cmd":"shutdown","token":"***"}
< {"ok":true,"reply":"shutting down"}
Python server exited: true, exit code 0
```

Four techniques in it are worth knowing in their own right.

- **A secret for this run only.** `SecureRandom` makes 16 unpredictable bytes, and the token reaches the server through the child's environment, never the command line, where other users could see it in a process list.
- **Starting another program.** `ProcessBuilder(python, script, "0", "--quiet")` describes the command and `start()` runs it; `0` asks for an ephemeral port, `--quiet` silences the per-client log.
- **Learning the port.** The demo reads the child's output until the line `LaundryStatusServer (python) ready on port N` and takes the last word. A daemon thread then keeps reading and discarding the output; otherwise the pipe would fill and block the server.
- **Stopping properly.** `shutdown` with the token, then `waitFor` with a ten-second limit; exit code 0 means a normal end.

Listing 15.22 works in the other direction: Python starts the **Java** server (compile it first with `javac *.java`) and runs the Python client's scenario.

Listing 15.22 `interop_demo.py`: start the Java server as a separate process and talk to it from Python.

```
"""interop_demo.py: start the JAVA server as a separate process, talk to it from PYTHON.

Compile the Java files first: cd ../java && javac *.java
"""
import os
import secrets
import shutil
import subprocess
import sys
import threading

from laundry_status_client import LaundryStatusClient, run_scenario, show


def main():
    class_dir = sys.argv[1] if len(sys.argv) > 1 else os.path.join("..", "java")
    java = shutil.which("java")
    if java is None:
        print("java was not found on the PATH")
        return

    token = secrets.token_hex(16)  # a fresh random token for this run only
    env = dict(os.environ, PROFITINA_STAFF_TOKEN=token)
    server = subprocess.Popen(
        [java, "-cp", class_dir, "LaundryStatusServer", "0", "--quiet"],
        stdout=subprocess.PIPE, stderr=subprocess.STDOUT, env=env,
        text=True, encoding="utf-8")

    port = None
    for line in server.stdout:
        if line.startswith("LaundryStatusServer"):  # "... ready on port 43127"
            port = int(line.split()[-1])
            break
    if port is None:
        print("the Java server did not start")
        return
    # Keep the pipe empty so the server never blocks on a full buffer.
    threading.Thread(target=server.stdout.read, daemon=True).start()
    print("started the Java server as a separate process")
```

Listing 15.22 (Continued) *interop_demo.py: start the Java server as a separate process and talk to it from Python.*

```
    with LaundryStatusClient("127.0.0.1", port) as client:
        print("answered by: " + client.ping()["server"])
        run_scenario(client, token)
        print("> " + show({"cmd": "shutdown", "token": token}))
        print("< " + show(client.shutdown_server(token)))
    try:
        code = server.wait(timeout=10)
        print(f"Java server exited: true, exit code {code}")
    except subprocess.TimeoutExpired:
        server.kill()
        print("Java server exited: false, exit code -1")


if __name__ == "__main__":
    main()
```

```
started the Java server as a separate process
answered by: java
> {"cmd":"ping"}
< {"ok":true,"reply":"pong","server":"java"}
> {"cmd":"status","order":"ORD-0713","phone_last4":"7890"}
< {"ok":true,"order":"ORD-0713","customer":"B***
S******","status":"WASHING","ready_by":"2026-09-24 16:00"}
> {"cmd":"status","order":"ORD-0713","phone_last4":"0000"}
< {"ok":false,"error":"no order with that number and phone"}
> {"cmd":"status","order":"ORD-9999","phone_last4":"7890"}
< {"ok":false,"error":"no order with that number and phone"}
> this is not JSON
< {"ok":false,"error":"malformed request"}
> {"cmd":"advance","order":"ORD-0713","token":"***"}
< {"ok":false,"error":"not authorized"}
> {"cmd":"advance","order":"ORD-0713","token":"***"}
< {"ok":true,"order":"ORD-0713","status":"DRYING"}
> {"cmd":"status","order":"ORD-0713","phone_last4":"7890"}
< {"ok":true,"order":"ORD-0713","customer":"B***
S******","status":"DRYING","ready_by":"2026-09-24 16:00"}
> {"cmd":"shutdown","token":"***"}
< {"ok":true,"reply":"shutting down"}
Java server exited: true, exit code 0
```

Apart from the words “Python” and “Java” and the `server` field of `ping`, the two outputs are identical: two independent implementations of **Table 15.4** give the same answer to every request, refusals included. That is what a specification buys: the Java client’s owner never read the Python server’s code, and did not need to.

15.7.6 The cost of a conversation

Every request in this protocol is a **round trip**: the request travels to the server, the server works, the response travels back. The time for one message of S bytes over a link with one-way delay (latency) L and bandwidth B is

$$T = L + \frac{S}{B} \tag{15.4}$$

and a conversation of k request–response pairs, each needing a server time t_s, takes about

$$T(k) \approx k \cdot (\mathrm{RTT} + t_s), \qquad \mathrm{RTT} = 2L + \frac{S_{\mathrm{req}} + S_{\mathrm{resp}}}{B}. \tag{15.5}$$

In numbers: a status request is about 60 bytes and its response about 120. Between the laundry and a data center 20 ms away over 10 Mbit/s, S/B for 180 bytes is $180 \times 8/10^7 \approx 0.14$ ms, negligible next to $2L = 40$ ms. By Equation (15.5), ten separate requests cost about 400 ms and one request for ten orders about 40 ms. Over a real network the number of round trips matters far more than the number of bytes, so prefer a few rich messages to many small ones; Exercise 15.6 practices the arithmetic.

15.8 HTTP: The Protocol of the Web

Our order-status protocol is private. The most widely spoken protocol in the world is **HTTP** (Hypertext Transfer Protocol), the language of every browser and of most services programs call over the Internet [13]. It is built as our protocols are, text lines on TCP, but it is standardized, so a browser, a phone app and a program in any language can use the same server.

15.8.1 What an HTTP conversation looks like

An HTTP request is a few lines of text: a first line with a **method** (what to do), a **path** (which resource) and the version; **header** lines `Name: value`; an empty line; and, for some methods, a body. The response has the same shape with a **status line** first. Here is a request for order ORD-0714 and its answer as they travel through the socket (each line ends with `\r\n`):

```
GET /orders/ORD-0714 HTTP/1.1
Host: 127.0.0.1
Connection: close

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 47

{"ok":true,"order":"ORD-0714","status":"READY"}
```

The common methods are `GET` (read), `POST` (create or submit), `PUT` (replace) and `DELETE`. The status code's first digit gives the class of the answer; **Table 15.5** lists the codes a beginner meets first.

Table 15.5 Common HTTP status codes. Codes 2xx mean success, 4xx a problem with the request, 5xx a problem in the server.

Code	Meaning	Laundry example
200 OK	The request succeeded; the body has the answer	status of an existing order
201 Created	A new resource was created	a new order was registered
400 Bad Request	The request is malformed	the body is not valid JSON
401 Unauthorized	Credentials are missing or wrong	staff command without a valid token
403 Forbidden	Credentials are valid but not sufficient	a courier tries to change prices
404 Not Found	No such resource	order ORD-9999
429 Too Many Requests	The client must slow down	too many guesses of phone digits
500 Internal Server Error	The server failed	an uncaught exception in the handler
503 Service Unavailable	The server is overloaded or in maintenance	during an update

HTTP standardizes exactly what we had to invent: which operation is meant (method and path), success or failure (status code), and metadata (headers such as `Content-Type`). Its framing: the headers end at an empty line, and `Content-Length` says how many body bytes follow.

15.8.2 An HTTP server and client in each language

Both standard libraries include a small HTTP server and an HTTP client. **Listing 15.23** starts a server inside the program on loopback and an ephemeral port, so no Internet is needed; asks it three questions with `java.net.http.HttpClient`; asks one more by typing HTTP by hand into a plain socket; and stops the server.

Listing 15.23 `HttpStatusDemo.java`: an HTTP server on localhost, queried with `HttpClient` and with a bare socket.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.time.Duration;
import java.util.Map;

/** A tiny HTTP server on localhost, queried with HttpClient and with a bare socket. */
public class HttpStatusDemo {
    private static final Map<String, String> STATUS = Map.of(
            "ORD-0713", "WASHING", "ORD-0714", "READY", "ORD-0715", "RECEIVED");

    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(
                new InetSocketAddress(InetAddress.getLoopbackAddress(), 0), 0);
        server.createContext("/health", exchange -> reply(exchange, 200,
                "text/plain; charset=utf-8", "ok"));
        server.createContext("/orders/", HttpStatusDemo::orderStatus);
        server.start();
        int port = server.getAddress().getPort();
        String base = "http://127.0.0.1:" + port;
        System.out.println("HTTP server started on an ephemeral port");

        HttpClient client = HttpClient.newBuilder()
                .proxy(HttpClient.Builder.NO_PROXY)  // talk directly, ignore any system
proxy
                .connectTimeout(Duration.ofSeconds(2))
                .build();
        for (String path : new String[] {"/health", "/orders/ORD-0714", "/orders/ORD-
9999"}) {
            HttpRequest request = HttpRequest.newBuilder(URI.create(base + path))
                    .timeout(Duration.ofSeconds(5))
                    .GET()
                    .build();
            HttpResponse<String> response = client.send(request,
                    HttpResponse.BodyHandlers.ofString(StandardCharsets.UTF_8));
```

Listing 15.23 (Continued) *HttpStatusDemo.java: an HTTP server on localhost, queried with HttpClient and with a bare socket.*

```
            System.out.println("GET " + path + " -> " + response.statusCode() + " "
                    + response.headers().firstValue("content-type").orElse("?"));
            System.out.println("    body: " + response.body());
        }

        // HTTP is only text over TCP: the same question, typed by hand on a socket.
        try (Socket socket = new Socket(InetAddress.getLoopbackAddress(), port);
             PrintWriter out = new PrintWriter(new OutputStreamWriter(
                     socket.getOutputStream(), StandardCharsets.US_ASCII));
             BufferedReader in = new BufferedReader(new InputStreamReader(
                     socket.getInputStream(), StandardCharsets.UTF_8))) {
            out.print("GET /health HTTP/1.1\r\nHost: 127.0.0.1\r\nConnection:
close\r\n\r\n");
            out.flush();
            System.out.println("raw socket, first line of the answer: " + in.readLine());
        }

        server.stop(0);  // stop at once; no requests are in progress
        System.out.println("HTTP server stopped");
    }

    private static void orderStatus(HttpExchange exchange) throws IOException {
        String id = exchange.getRequestURI().getPath().substring("/orders/".length());
        String status = STATUS.get(id);
        if (status == null) {
            reply(exchange, 404, "application/json; charset=utf-8",
                    Json.write(Json.object("ok", false, "error", "no such order")));
        } else {
            reply(exchange, 200, "application/json; charset=utf-8",
                    Json.write(Json.object("ok", true, "order", id, "status", status)));
        }
    }

    private static void reply(HttpExchange exchange, int code, String type, String body)
            throws IOException {
        byte[] bytes = body.getBytes(StandardCharsets.UTF_8);
        exchange.getResponseHeaders().set("Content-Type", type);
        exchange.sendResponseHeaders(code, bytes.length);
        try (OutputStream os = exchange.getResponseBody()) {
            os.write(bytes);
        }
    }
}
```

```
HTTP server started on an ephemeral port
GET /health -> 200 text/plain; charset=utf-8
    body: ok
GET /orders/ORD-0714 -> 200 application/json; charset=utf-8
    body: {"ok":true,"order":"ORD-0714","status":"READY"}
GET /orders/ORD-9999 -> 404 application/json; charset=utf-8
    body: {"ok":false,"error":"no such order"}
raw socket, first line of the answer: HTTP/1.1 200 OK
HTTP server stopped
```

- `HttpServer.create` comes from `com.sun.net.httpserver`, part of every JDK. `createContext(path, handler)` connects a path prefix to a handler that receives an `HttpExchange`, one request and its response.
- `reply` sets the headers, sends the status code and the body length in *bytes* with `sendResponseHeaders`, then writes the body.
- `NO_PROXY` makes the client connect directly rather than through a proxy configured on the computer, which could not reach loopback. `client.send(request,`

`BodyHandlers.ofString(UTF_8))` waits and returns an `HttpResponse` with `statusCode()`, `headers()` and `body()`; a 404 is a response, not an exception.

- The last block proves that HTTP is text over TCP: an ordinary `Socket`, three header lines and an empty line written by hand, and the status line `HTTP/1.1 200 OK` comes back. `server.stop(0)` stops the server at once.

Listing 15.24 is the Python version, with `http.server` and `urllib.request`.

Listing 15.24 `http_status_demo.py`: the same server and questions in Python.

```
"""http_status_demo.py: a tiny HTTP server on localhost, queried with urllib and a bare
socket."""
import json
import socket
import threading
import urllib.error
import urllib.request
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

STATUS = {"ORD-0713": "WASHING", "ORD-0714": "READY", "ORD-0715": "RECEIVED"}


def write(value):
    return json.dumps(value, separators=(",", ":"), ensure_ascii=False)


class StatusHandler(BaseHTTPRequestHandler):
    protocol_version = "HTTP/1.1"

    def do_GET(self):
        if self.path == "/health":
            self._reply(200, "text/plain; charset=utf-8", "ok")
        elif self.path.startswith("/orders/"):
            order_id = self.path[len("/orders/"):]
            status = STATUS.get(order_id)
            if status is None:
                self._reply(404, "application/json; charset=utf-8",
                            write({"ok": False, "error": "no such order"}))
            else:
                self._reply(200, "application/json; charset=utf-8",
                            write({"ok": True, "order": order_id, "status": status}))
        else:
            self._reply(404, "text/plain; charset=utf-8", "not found")

    def _reply(self, code, content_type, body):
        data = body.encode("utf-8")
        self.send_response(code)
        self.send_header("Content-Type", content_type)
        self.send_header("Content-Length", str(len(data)))
        self.end_headers()
        self.wfile.write(data)

    def log_message(self, format, *args):
        pass  # keep the console quiet
```

Listing 15.24 (Continued) *http_status_demo.py: the same server and questions in Python.*

```
def main():
    server = ThreadingHTTPServer(("127.0.0.1", 0), StatusHandler)
    port = server.server_address[1]
    threading.Thread(target=server.serve_forever, daemon=True).start()
    base = f"http://127.0.0.1:{port}"
    print("HTTP server started on an ephemeral port")

    # talk directly, ignore any system proxy
    opener = urllib.request.build_opener(urllib.request.ProxyHandler({}))
    for path in ["/health", "/orders/ORD-0714", "/orders/ORD-9999"]:
        try:
            response = opener.open(base + path, timeout=5)
        except urllib.error.HTTPError as e:  # 4xx and 5xx arrive as an exception
            response = e
        with response:
            body = response.read().decode("utf-8")
            print(f"GET {path} -> {response.status} {response.headers['Content-Type']}")
            print(f"    body: {body}")

    # HTTP is only text over TCP: the same question, typed by hand on a socket.
    with socket.create_connection(("127.0.0.1", port), timeout=5) as sock:
        sock.sendall(b"GET /health HTTP/1.1\r\nHost: 127.0.0.1\r\nConnection:
close\r\n\r\n")
        with sock.makefile("r", encoding="utf-8", newline="\r\n") as reader:
            first = reader.readline().rstrip("\r\n")
    print("raw socket, first line of the answer: " + first)

    server.shutdown()      # stop serve_forever()
    server.server_close()  # release the port
    print("HTTP server stopped")


if __name__ == "__main__":
    main()
```

```
HTTP server started on an ephemeral port
GET /health -> 200 text/plain; charset=utf-8
    body: ok
GET /orders/ORD-0714 -> 200 application/json; charset=utf-8
    body: {"ok":true,"order":"ORD-0714","status":"READY"}
GET /orders/ORD-9999 -> 404 application/json; charset=utf-8
    body: {"ok":false,"error":"no such order"}
raw socket, first line of the answer: HTTP/1.1 200 OK
HTTP server stopped
```

The Python handler inherits from `BaseHTTPRequestHandler` and overrides `do_GET`: the template method pattern of Chapter 7, in which the library parses the request and calls our method. `ThreadingHTTPServer` serves each request on its own thread until `shutdown()`. `urllib` reports 4xx and 5xx by raising `HTTPError`, which is also a response, so the program prints it like any other; `ProxyHandler({})` is Python's `NO_PROXY`.

A service for a phone app would speak HTTP through a web framework (Profitina's back end uses FastAPI, Section 15.12). Raw socket protocols remain worth knowing: they are what HTTP is made of, games and chats use them when one-request-one-response is too rigid, and they show what a framework does for you.

15.9 Security and Personal Data

A program that listens on a network accepts input from strangers, some of whom may be trying to break it. And a service that handles customers' names and phone numbers is subject to

Indonesia's Law No. 27 of 2022 on Personal Data Protection, *Undang-Undang Pelindungan Data Pribadi* (UU PDP) [14], fully in force since October 2024. This section is not legal advice; it shows how the rules become ordinary programming decisions, most of which our service already makes.

15.9.1 Seven habits for networked code

1. **Listen only where you must.** Our servers bind to `127.0.0.1`. To let the counter laptop reach the back-room server, bind to that server's private address or to all addresses (`0.0.0.0`), and let the firewall decide who may connect. Open a service to the Internet only behind proper protection.
2. **Trust nothing from the other side.** Check every field's existence and type; answer a malformed request with a refusal, never a crash.
3. **Limit everything**: line length (1024 characters), silence (60 seconds), workers (16). Without limits, one hostile client can exhaust memory or threads for everyone, a *denial of service*.
4. **Keep secrets out of code and logs.** Environment variable, constant-time comparison, printed as `***`. A secret ever committed to Git is public forever and must be replaced.
5. **Encrypt anything that leaves the machine.** Plain TCP is readable by anyone on the same Wi-Fi. The remedy is **TLS**, which turns HTTP into HTTPS [15]: `SSLSocket`/`SSLServerSocket` in Java (automatic for `https://` in `HttpClient`), the `ssl` module in Python. In production a reverse proxy or tunnel usually provides it in front of the application.
6. **Say little when refusing.** Separate messages for "no such order" and "wrong phone" would let a guesser find the existing orders first.
7. **Slow down guessers.** Four phone digits allow 10^4 values; guessing in random order takes on average about half of them, so at r attempts per second the expected time is

$$t_{\text{guess}} \approx \frac{10^4 + 1}{2 \cdot r}. \tag{15.6}$$

At $r = 1000$, Equation (15.6) gives about five seconds, which is why real services add **rate limiting**. Allowing five attempts per minute for one order, $r = 1/12$ per second, gives $5000.5 \times 12 \approx 60\,000$ seconds, about 17 hours, long enough for staff to notice. Exercise 15.10 designs it.

15.9.2 What UU PDP asks of a program like ours

UU PDP distinguishes the **data subject** (the customer), the **data controller** (Profitina Laundry, which decides why and how data is processed) and the **data processor** (for example a hosting company acting for it). Personal data is anything that identifies a person, alone or combined: a name, a phone number, even an order history. **Table 15.6** maps the main obligations onto our service.

Table 15.6 The main obligations of UU PDP No. 27/2022, and how each appears in a small networked service.

Obligation (in plain words)	What it means for the order-status service
Process data only on a lawful basis, such as consent or the need to perform a contract	Collect the name and phone number because the customer ordered a service; tell them why at the counter
Collect only what the purpose needs (data minimization)	The status service needs the order number and the last four phone digits; it does not need the address
Keep data accurate, and let the subject see, correct and delete it	Staff tools must support correcting and deleting a customer's record
Protect data with appropriate technical measures	Bind to loopback or a private address, TLS on the network, secrets in the environment, masked names in responses
Do not reveal more than necessary	Show initials only; never return the phone number; the same refusal for all failures
Keep data no longer than needed	Delete or anonymize orders some months after pickup
Report a failure of protection promptly	UU PDP requires written notice to the data subjects and to the supervising agency within 3 × 24 hours of a data protection failure
Keep a record of processing	Logs that say *what* happened (a status request succeeded) without copying *whose* data

Two lines of the table are visible in our code. The log records `status -> ok`, not the order or the name, because logs are copied and read by many people. And `publicView` returns initials and never the phone number, because the purpose is to say whether laundry is ready, not to tell whoever holds a receipt who the customer is. Data protection is a set of small decisions made while designing the protocol, one more reason to design it on paper first.

> ⚠ **WARNING.** Never use real customers' data in a team project, a test, a screenshot or a demonstration. Invent names and numbers, as this chapter does. Test data that is real personal data falls under the same law as production data.

15.10 Testing Networked Programs

Networked code needs the automated tests of Chapter 6 more than most, because its bugs appear only in particular timings and failures. Four techniques make it manageable.

- **Test the protocol without the network**: `processInput` and `handle` do no I/O.
- **Start a real server inside the test, on port 0**, so it never collides with another test or server; `start` returns the port and `stop` releases it.
- **Use a fresh server for every test**, so that no test sees data changed by another.
- **Test the failures on purpose**: junk, floods, many clients at once, a stopped server. The happy path works first; the real bugs live elsewhere.

Listing 15.25 applies all four. Following Chapter 6, it is a self-contained checker with a `main`, needing only the JDK; in JUnit 5, `@BeforeEach` and `@AfterEach` would start and stop the server.

Listing 15.25 `LaundryStatusCheck.java`: ten automated checks, each against its own server on an ephemeral port.

```
import java.io.IOException;
import java.net.ConnectException;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

/**
 * Automated checks for the order-status service. Every check starts its own server on an
 * ephemeral port, talks to it over real sockets, and stops it again.
 */
public class LaundryStatusCheck {
    private static final String TOKEN = "test-token-not-a-secret";
    private static int passed = 0;
    private static int failed = 0;

    interface Check {
        void run(int port) throws Exception;
    }

    public static void main(String[] args) {
        check("ping answers pong", port -> {
            try (LaundryStatusClient c = new LaundryStatusClient("127.0.0.1", port)) {
                expect("pong", c.ping().get("reply"));
            }
        });
        check("status shows initials only, never the phone", port -> {
            try (LaundryStatusClient c = new LaundryStatusClient("127.0.0.1", port)) {
                Map<String, Object> r = c.status("ORD-0714", "5432");
                expect("S*** W********", r.get("customer"));
                expect("READY", r.get("status"));
                expect(false, r.containsKey("phone"));
            }
        });
        check("wrong phone and unknown order look the same", port -> {
            try (LaundryStatusClient c = new LaundryStatusClient("127.0.0.1", port)) {
                expect(c.status("ORD-0714", "0000"), c.status("ORD-9999", "5432"));
            }
        });
        check("junk is refused and the connection survives", port -> {
            try (LaundryStatusClient c = new LaundryStatusClient("127.0.0.1", port)) {
                expect("malformed request", c.sendRaw("{not json").get("error"));
                expect("malformed request", c.sendRaw("[1, 2, 3]").get("error"));
                expect("unknown command: fly", c.send(Json.object("cmd",
"fly")).get("error"));
```

Listing 15.25 (Continued) *LaundryStatusCheck.java: ten automated checks, each against its own server on an ephemeral port.*

```
                expect(true, c.ping().get("ok"));
            }
        });
        check("advance needs the staff token", port -> {
            try (LaundryStatusClient c = new LaundryStatusClient("127.0.0.1", port)) {
                expect("not authorized", c.advance("ORD-0715", "wrong").get("error"));
                expect("RECEIVED", c.status("ORD-0715", "2222").get("status"));
            }
        });
        check("advance walks the status forward and stops at PICKED_UP", port -> {
            try (LaundryStatusClient c = new LaundryStatusClient("127.0.0.1", port)) {
                expect("PICKED_UP", c.advance("ORD-0714", TOKEN).get("status"));
                expect("order already picked up", c.advance("ORD-0714",
TOKEN).get("error"));
            }
        });
        check("a request longer than 1024 characters is refused", port -> {
            try (LaundryStatusClient c = new LaundryStatusClient("127.0.0.1", port)) {
                expect("request too long", c.sendRaw("x".repeat(5000)).get("error"));
            }
        });
        check("twenty clients at once are all answered", port -> {
            ExecutorService pool = Executors.newFixedThreadPool(20);
            List<Future<Object>> answers = new ArrayList<>();
            for (int i = 0; i < 20; i++) {
                answers.add(pool.submit(() -> {
                    try (LaundryStatusClient c = new LaundryStatusClient("127.0.0.1",
port)) {
                        return c.status("ORD-0713", "7890").get("status");
                    }
                }));
            }
            for (Future<Object> answer : answers) {
                expect("WASHING", answer.get());
            }
            pool.shutdown();
        });
        check("five staff clients advancing at once lose no update", port -> {
            ExecutorService pool = Executors.newFixedThreadPool(5);
            List<Future<Object>> answers = new ArrayList<>();
            for (int i = 0; i < 5; i++) {  // RECEIVED + 5 steps = PICKED_UP, exactly
                answers.add(pool.submit(() -> {
                    try (LaundryStatusClient c = new LaundryStatusClient("127.0.0.1",
port)) {
                        return c.advance("ORD-0715", TOKEN).get("ok");
                    }
                }));
            }
            for (Future<Object> answer : answers) {
```

Listing 15.25 (Continued) *LaundryStatusCheck.java: ten automated checks, each against its own server on an ephemeral port.*

```
                expect(true, answer.get());
            }
            pool.shutdown();
            try (LaundryStatusClient c = new LaundryStatusClient("127.0.0.1", port)) {
                expect("PICKED_UP", c.status("ORD-0715", "2222").get("status"));
            }
        });

        // Server down: once stopped, a client must get a clear error, not hang.
        LaundryStatusServer server = LaundryStatusServer.withSampleOrders(TOKEN, false);
        try {
            int port = server.start(0);
            server.stop();
            try {
                new LaundryStatusClient("127.0.0.1", port).close();
                record("a stopped server gives ConnectException", false);
            } catch (ConnectException e) {
                record("a stopped server gives ConnectException", true);
            }
        } catch (IOException e) {
            record("a stopped server gives ConnectException", false);
        }

        System.out.println((passed + failed) + " checks, " + passed + " passed, "
                + failed + " failed");
    }

    private static void check(String name, Check body) {
        LaundryStatusServer server = LaundryStatusServer.withSampleOrders(TOKEN, false);
        try {
            int port = server.start(0);  // a fresh server with fresh data for every check
            body.run(port);
            record(name, true);
        } catch (Throwable t) {
            record(name + " (" + t.getMessage() + ")", false);
        } finally {
            server.stop();
        }
    }

    private static void expect(Object expected, Object actual) {
        if (!expected.equals(actual)) {
            throw new AssertionError("expected " + expected + " but got " + actual);
        }
    }
```

Listing 15.25 (Continued) *LaundryStatusCheck.java: ten automated checks, each against its own server on an ephemeral port.*

```
    private static void record(String name, boolean ok) {
        if (ok) {
            passed++;
        } else {
            failed++;
        }
        System.out.println((ok ? "PASS  " : "FAIL  ") + name);
    }
}
```

```
PASS  ping answers pong
PASS  status shows initials only, never the phone
PASS  wrong phone and unknown order look the same
PASS  junk is refused and the connection survives
PASS  advance needs the staff token
PASS  advance walks the status forward and stops at PICKED_UP
PASS  a request longer than 1024 characters is refused
PASS  twenty clients at once are all answered
PASS  five staff clients advancing at once lose no update
PASS  a stopped server gives ConnectException
10 checks, 10 passed, 0 failed
```

The run takes a fraction of a second. “Twenty clients at once” checks that a pool and accept loop under load answer everyone. “Five staff clients advancing at once” tests Chapter 14’s lesson: from `RECEIVED`, exactly five steps reach `PICKED_UP`, so all five must succeed and the order must end `PICKED_UP`. Without `synchronized` on `OrderBook.advance`, two threads could read the same old status and write the same new one; a step would vanish and the order would end `READY`. (A test like this catches races sometimes; the proof of their absence is the `synchronized` keyword and Chapter 14’s reasoning.)

The last check is the team project’s “server down” case, automated: after `stop`, a client must get `ConnectException` at once, not hang.

Listing 15.26 is the same suite with `unittest`, whose `setUp` and `tearDown` start and stop a fresh server around each test.

Listing 15.26 `test_laundry_status.py`: the same ten tests with `unittest`.

```
"""test_laundry_status.py: automated tests for the order-status service.

Every test starts its own server on an ephemeral port, talks to it over real
sockets, and stops it again. Run: python3 test_laundry_status.py
"""
import unittest
from concurrent.futures import ThreadPoolExecutor

from laundry_status_client import LaundryStatusClient
from laundry_status_server import LaundryStatusServer, OrderBook

TOKEN = "test-token-not-a-secret"


class LaundryStatusTest(unittest.TestCase):
    def setUp(self):
        # a fresh server with fresh data for every test
        self.server = LaundryStatusServer(OrderBook.sample(), TOKEN, verbose=False)
        self.port = self.server.start(0)

    def tearDown(self):
        self.server.stop()

    def client(self):
        return LaundryStatusClient("127.0.0.1", self.port)

    def test_ping_answers_pong(self):
        with self.client() as c:
            self.assertEqual("pong", c.ping()["reply"])

    def test_status_shows_initials_only(self):
        with self.client() as c:
            r = c.status("ORD-0714", "5432")
            self.assertEqual("S*** W********", r["customer"])
            self.assertEqual("READY", r["status"])
            self.assertNotIn("phone", r)

    def test_wrong_phone_and_unknown_order_look_the_same(self):
        with self.client() as c:
            self.assertEqual(c.status("ORD-0714", "0000"), c.status("ORD-9999", "5432"))
```

Listing 15.26 (Continued) *test_laundry_status.py: the same ten tests with unittest.*

```
    def test_junk_is_refused_and_connection_survives(self):
        with self.client() as c:
            self.assertEqual("malformed request", c.send_raw("{not json")["error"])
            self.assertEqual("malformed request", c.send_raw("[1, 2, 3]")["error"])
            self.assertEqual("unknown command: fly", c.send({"cmd": "fly"})["error"])
            self.assertTrue(c.ping()["ok"])

    def test_advance_needs_the_staff_token(self):
        with self.client() as c:
            self.assertEqual("not authorized", c.advance("ORD-0715", "wrong")["error"])
            self.assertEqual("RECEIVED", c.status("ORD-0715", "2222")["status"])

    def test_advance_stops_at_picked_up(self):
        with self.client() as c:
            self.assertEqual("PICKED_UP", c.advance("ORD-0714", TOKEN)["status"])
            self.assertEqual("order already picked up", c.advance("ORD-0714",
TOKEN)["error"])

    def test_long_request_is_refused(self):
        with self.client() as c:
            self.assertEqual("request too long", c.send_raw("x" * 5000)["error"])

    def test_twenty_clients_at_once(self):
        def ask(_):
            with self.client() as c:
                return c.status("ORD-0713", "7890")["status"]
        with ThreadPoolExecutor(max_workers=20) as pool:
            self.assertEqual(["WASHING"] * 20, list(pool.map(ask, range(20))))

    def test_five_staff_clients_lose_no_update(self):
        def advance(_):
            with self.client() as c:
                return c.advance("ORD-0715", TOKEN)["ok"]
        with ThreadPoolExecutor(max_workers=5) as pool:  # RECEIVED + 5 steps = PICKED_UP
            self.assertEqual([True] * 5, list(pool.map(advance, range(5))))
        with self.client() as c:
            self.assertEqual("PICKED_UP", c.status("ORD-0715", "2222")["status"])

    def test_stopped_server_refuses_connections(self):
        self.server.stop()
        with self.assertRaises(ConnectionRefusedError):
            self.client()


if __name__ == "__main__":
    unittest.main(verbosity=2)
```

```
test_advance_needs_the_staff_token
(__main__.LaundryStatusTest.test_advance_needs_the_staff_token) ... ok
test_advance_stops_at_picked_up
(__main__.LaundryStatusTest.test_advance_stops_at_picked_up) ... ok
test_five_staff_clients_lose_no_update
(__main__.LaundryStatusTest.test_five_staff_clients_lose_no_update) ... ok
test_junk_is_refused_and_connection_survives
(__main__.LaundryStatusTest.test_junk_is_refused_and_connection_survives) ... ok
test_long_request_is_refused (__main__.LaundryStatusTest.test_long_request_is_refused) ...
ok
test_ping_answers_pong (__main__.LaundryStatusTest.test_ping_answers_pong) ... ok
test_status_shows_initials_only
(__main__.LaundryStatusTest.test_status_shows_initials_only) ... ok
test_stopped_server_refuses_connections
(__main__.LaundryStatusTest.test_stopped_server_refuses_connections) ... ok
test_twenty_clients_at_once (__main__.LaundryStatusTest.test_twenty_clients_at_once) ... ok
test_wrong_phone_and_unknown_order_look_the_same
(__main__.LaundryStatusTest.test_wrong_phone_and_unknown_order_look_the_same) ... ok

----------------------------------------------------------------------
Ran 10 tests in 2.049s

OK
```

The Python suite takes about two seconds because each `stop()` waits for the accept loop's next 0.2-second timeout, a fair price for a server that always stops cleanly.

15.11 The Team Project

Everything in this book so far could be done by one person. The client–server application is where that stops, and where object-oriented design pays its biggest dividend: classes with clear interfaces let several people build one program without stepping on each other. This section is a practical guide for a team of two or three building a GUI-and-socket application, from the first meeting to the presentation; Section 15.11.10 gives a complete sample brief.

15.11.1 Choosing a topic

Good team projects share one property: the topic *needs* a client and a server, because several people use it at the same time from different places and see each other's actions. Examples:

- a **game with chat**, in which the server keeps the game state and the rules, and each client shows the board and sends moves;
- a small **social network** ("what the laundry is doing today"), in which clients post and read, and the server stores and distributes;
- a **collaboration tool**, such as a shared to-do board for a student organization or a shared shopping list for a boarding house;
- an **interactive portal**, such as a queue display for the laundry, a campus lost-and-found board or a menu-and-order board for a food stall;
- a small **startup idea** of your own, or anything else your team finds genuinely interesting.

Decide the topic and a working title in the first meeting, with a one-paragraph pitch: who uses it, what for, and why it needs a network. Then apply three tests. *Network*: if it works as well as a single program, it is not a client–server topic. *Size*: can the core be demonstrated after two thirds of the time? If not, cut it now. *Failure*: can you name something that goes wrong across the network (a client disconnects mid-game, the server restarts during a post) that matters to the user? Then you have something real to design and to show.

15.11.2 Splitting the work

A client–server application with a graphical interface falls naturally into five components, shown in **Figure 15.8**:

- the **server**, which accepts connections, runs the protocol and owns the shared state;
- the **client**, which connects, sends requests and interprets responses (the proxy of Section 15.7.4);
- **database and database connectivity**, which stores the state permanently, so that a restarted server remembers its orders;
- the **GUI**, the Swing or Tkinter window of Chapters 10 and 11, which the user actually sees;
- **testing and documentation**, which is a component in its own right and not an afterthought.

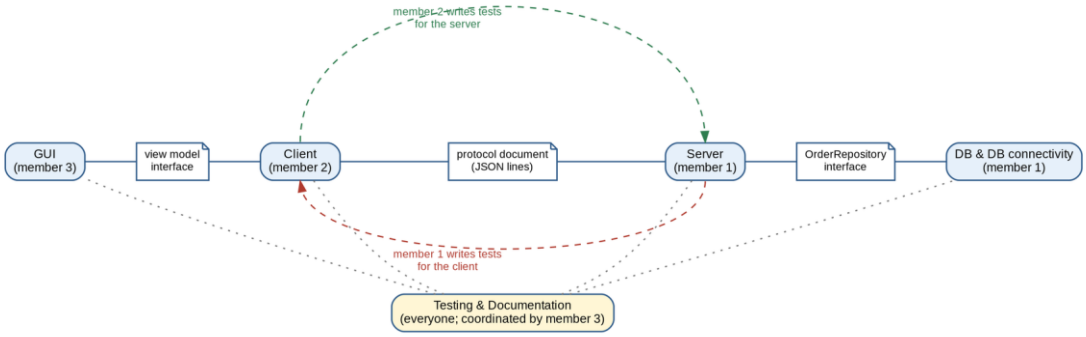


Figure 15.8 The five components of a team project, their owners, and the three interface documents between them. Dashed arrows show cross-testing: each owner writes tests for the component that uses theirs, or that theirs uses.

Each member owns one or more components and writes a short **specification and job description**: what the component does, offers, needs, and what will be delivered when. Ownership means "this person answers for it and reviews every change", not "nobody else may touch it".

The boundaries matter more than the components. **Figure 15.8** has three, each written down as an **interface document** before any code:

- between client and server, the **protocol document**: a table like **Table 15.4** plus a state diagram like **Figure 15.5**;
- between server and database, a **repository interface**: for example a Java `interface OrderRepository` with `find(id)`, `save(order)`, `all()`, or the equivalent Python abstract class;
- between client and GUI, a **view model**: the methods the window calls (`refresh()`, `advance(orderId)`) and the events it receives (`statusChanged`).

"Interface" is meant generally: not necessarily a Java `interface`, but any written agreement about what one part offers another, whether a table, a Markdown file or method signatures with Javadoc or docstrings. It must be written, read by both sides, and changed only by agreement. MVC (Chapter 11) is the same idea inside one program; here it applies between people.

15.11.3 From specification to design to implementation

Carry the project through three visible phases, keeping each one's products in the repository.

1. **Specification**: *what* the system does for the user, as one-sentence features ("a customer can see an order's status by giving its number and the last four phone digits"), each with what counts as working. The protocol document belongs here.

2. **Design**: *how* the parts fit: components diagram, interface documents, a UML class diagram per component (Chapter 7), and the protocol's state diagrams.
3. **Implementation**: the code, written against the design, with the tests.

The phases overlap, since implementation teaches you things, but each should *exist* before the next begins; a team that codes on day one produces three programs that almost fit. An order of work that uses three people from the first week:

- Week 1: all three together write the specification and the interface documents, and set up the repository.
- Week 2: the server owner writes the server, testing it with a scripted client; the client owner writes the proxy, testing it against a **stub server**, a tiny fake that gives fixed, correct answers; the GUI owner builds the window on a stub client. Parallel work is possible only because the interfaces are fixed.
- Week 3: replace the stubs by the real parts, one boundary at a time, and run the cross tests of the next section.
- Week 4: failure handling, documentation, rehearsal.

15.11.4 Test cases across the interface

Think about test cases early and *across* each boundary. When A uses B's functions, A's owner knows best what A needs from B, so A's owner designs the test cases for B, and B's owner designs those for A. In **Figure 15.8**, the client owner tests the server and the server owner tests the client.

The reason is human: the person who writes a component tests what they intended; the person who uses it tests what they assumed. Bugs live in the gap, and cross-testing finds it before demonstration day. Cross tests live in the repository, like **Listing 15.25**, and run on every change. A useful test plan contains:

- **unit tests** of each class, as in Chapter 6, including the protocol object without the network;
- **stub-based tests**, in which a fake client tests the real server, or a fake server tests the real client;
- **multithreaded tests**, in which several clients use the server at once, as in **Listing 15.13** and the two concurrency checks of **Listing 15.25**;
- **failure tests**: malformed input, a server that is down, a client that vanishes;
- **distributed tests**, with the server on one laptop and clients on others over a shared network, at least once before the presentation; loopback hides firewalls, wrong bind addresses and slow Wi-Fi.

15.11.5 Working together with Git

Passing files around in chat loses work. **Git** records every change, who made it and why, and lets several people work on separate copies and combine them safely [16]; a shared repository on GitHub or GitLab holds the true version of the project.

The workflow that suits a small team is *feature branches* (**Figure 15.9**). `main` always compiles and passes its tests. Each piece of work gets its own branch; when it is ready, its owner opens a **pull request**, a teammate reviews it (reads, runs, approves or asks for changes), and it is merged.

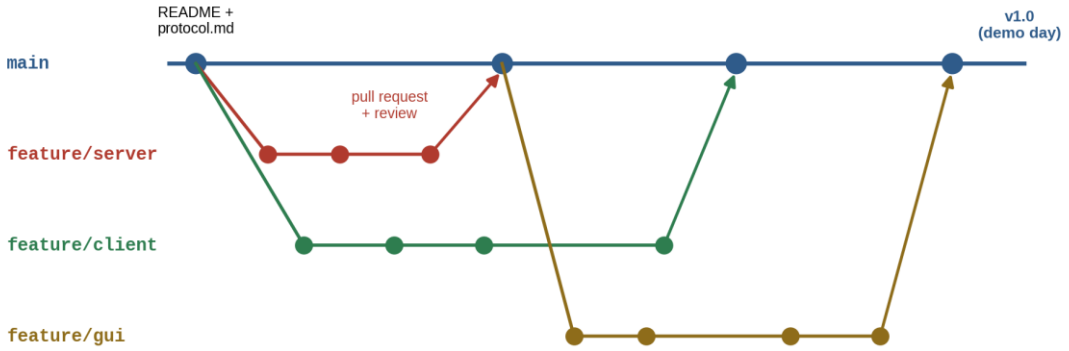


Figure 15.9 Feature branches. The team starts `main` with the README and the protocol document; each member works on a branch; every branch returns to `main` through a reviewed pull request; the demonstrated version is tagged.

Table 15.7 lists the first week's commands; VS Code's *Source Control* view offers them too.

Table 15.7 The Git commands a team member needs in the first week.

Command	What it does
`git clone <url>`	copy the shared repository to your computer (once)
`git switch -c feature/server`	create a new branch and move to it
`git status`	show which files changed and which are staged
`git add <file>`	stage a change for the next commit
`git commit -m "Add status command"`	record the staged changes with a message
`git push -u origin feature/server`	send your branch to the shared repository
`git switch main` then `git pull`	bring your copy of `main` up to date
`git merge main` (on your branch)	bring the latest `main` into your branch
`git log --oneline --graph`	show the history as a picture like **Figure 15.9**

Habits that prevent most trouble:

- **Commit small and often**, each commit one logical change with a message saying why ("Refuse lines longer than 1024 characters").
- **Pull before you start and merge `main` into your branch often**, so conflicts stay small.
- **Resolve conflicts together.** Git marks both versions between `<<<<<<<`, `=======` and `>>>>>>>`; agree which is right, edit, `git add`, `git commit` (Exercise 15.13).
- **Keep secrets and junk out** with a `.gitignore` (`*.class`, `__pycache__/`, editor folders, files with tokens).
- **Keep `main` green.** Never merge a branch that does not compile or whose tests fail.

15.11.6 The README

Every repository starts with a `README.md`, the first thing anyone sees. It answers, in order: what is this, how do I run it, how do I use it, where is everything else. A template for a client–server project:

```
# Laundry Live Board

A queue board for Profitina Laundry: the counter registers orders, staff move them
through washing, drying and ironing, and every screen in the shop shows the queue live.

## Team
- Ayu (server, database)    - Bagus (client, protocol)    - Citra (GUI, testing lead)

## Requirements
Java 21 or later (Temurin); Python 3.10 or later for the test scripts.

## Build and run
    cd server && javac *.java && java BoardServer 5060
    cd client && javac *.java && java BoardClient localhost 5060
Set PROFITINA_STAFF_TOKEN in the environment first (see docs/installation.md).

## Using it
Screenshots and a two-minute walk-through: docs/user-guide.md

## Documentation
docs/protocol.md  docs/design.md  docs/test-plan.md  docs/troubleshooting.md

## Tests
    cd server && java BoardServerCheck
```

15.11.7 Failure: the server goes down, a client goes down

An application that works only when both sides are healthy and the network is perfect is not finished. Two failures must be handled deliberately and demonstrated:

- **When the server goes down, the client must know.** No hang, no stack trace: a plain message ("the board server is not reachable; retrying"), the user's unsent input kept, retries with growing waits (Exercise 15.7), and a clear message on giving up. Every blocking call needs a timeout.
- **When a client goes down, the server must know.** A client that disconnects or crashes must not crash the server, leak its thread or socket, or affect others; end-of-stream, exceptions and idle timeouts all lead to the `finally` block that cleans up, and in a chat or game the others are told.

Both rely on **exception handling** (Chapter 5) for connection errors on either side. Write the handlers first, and test both failures by stopping the server mid-session and killing a client mid-request.

15.11.8 Documentation that people use

Alongside the code, write four short documents for four readers:

- a **user guide**, for the person who uses the application: what it does, with screenshots and a short walk-through;
- **performance criteria**, for the team and its reviewers: what "working well" means, in numbers you can check, for example "20 clients at once, every status answer within 200 ms on the campus network", and how you measured it;
- an **installation guide**, for the person who sets it up: exact steps on Windows, macOS and Linux, the ports it uses, the environment variables it needs;
- a **troubleshooting guide**, for the person for whom it does not work: the error messages the program can print, and what each one means ("Address already in use: another program uses port 5060; stop it or start the server with another port").

15.11.9 Presenting the project

A presentation is a demonstration with a story; plan it like a protocol.

- **Open with the problem**, in one sentence, from the user's point of view: "Customers phone the counter all day to ask if their laundry is ready."
- **Demonstrate live, on more than one computer**, clients seeing each other's actions. Then stop the server on purpose and let the audience watch the client report it and recover: nothing convinces like a failure handled well.
- **Show the design briefly**: the components diagram, the protocol table, one interesting class. Explain one decision you are proud of and one you would change.
- **Show the tests running**, including a concurrency test.
- **Let every member speak** about the part they own; a reviewer will want to hear each person explain their own code.
- **Prepare for bad luck**: rehearse twice on the actual machines, test the room's network early, and keep a screen recording of a good run as a backup.
- **Close honestly**: what works, what does not yet, and what you learned.

15.11.10 A complete sample project brief

This brief is written the way a team should write its own in the first week, small enough for three people and four weeks.

Title. Laundry Live Board.

Pitch. Profitina Laundry has one counter, four washers, three dryers and a stream of customers who want to know when their laundry will be ready. The Live Board is a client–server application: staff at the counter register orders and move them through the stages; every screen in the shop (the counter laptop, a monitor facing the waiting area, the staff tablet) shows the queue and updates within a second of any change; a customer can check an order's status at a self-service screen with the order number and the last four digits of their phone.

Users and features.

- F1. Counter staff register an order: customer name, phone, services, weight. The server assigns the order number.
- F2. Staff move an order to its next stage; all screens update within one second.
- F3. The waiting-area monitor shows the queue: order number, masked name, stage, estimated ready time. No phone numbers ever appear.
- F4. A customer checks one order at the self-service screen with the order number and phone digits; after five failed attempts that screen waits one minute.
- F5. A staff chat line for short messages ("W2 is free").
- F6. If the server restarts, all orders are still there; clients reconnect by themselves.

Components and owners. Ayu: server and database (SQLite through JDBC, behind an `OrderRepository` interface). Bagus: client library and protocol, including reconnection. Citra: Swing GUI for the counter and the monitor, testing lead, documentation lead.

Protocol. JSON lines over TCP as in Section 15.7, with `register`, `advance`, `status`, `queue`, `chat` and `bye`, plus one addition: after `{"cmd":"subscribe"}`, the server *pushes* an `{"event":"changed",...}` line whenever the queue changes. The client then needs a listener

thread that reads events while the GUI sends requests, like the chat members of Solution 15.11. The protocol document has a table like **Table 15.4** and a state diagram of a client session.

Milestones. **Table 15.8** gives one deliverable per week, each checked by someone other than the person who produced it.

Table 15.8 The milestone plan of the sample project, one deliverable per week, each checked by someone other than the person who produced it.

Week	Deliverable	Checked by
1	Specification, protocol document, components diagram, interface documents, repository with README and `.gitignore`	the whole team, together
2	Server passes its unit tests with a scripted client; client library passes its tests against a stub server; GUI runs against a stub client	the owner of the neighboring component (cross tests)
3	Real parts joined; F1–F4 work on one computer; concurrency test with 20 clients passes	Citra runs the full test plan
4	F5–F6; failure handling demonstrated on three computers; user, installation, performance and troubleshooting documents; two rehearsals	everyone

Performance criteria. 20 simultaneous clients; a change visible on every screen within one second on the shop's Wi-Fi; a status answer within 200 ms; server memory below 256 MB.

Personal data. Test data only, invented names; phone numbers never displayed or logged; masked names on public screens; orders deleted 90 days after pickup; the staff token in the environment.

Risks. Push updates and the GUI thread (mitigation: the Swing rules of Chapter 10, updates through `SwingUtilities.invokeLater`); Wi-Fi at the presentation (mitigation: a spare router and a recording).

15.11.11 A checklist for the team

Run through this list together a week before the presentation and again the day before. It is not a score sheet; it is what a careful reviewer, or a real customer, will ask.

- Can each of us state the topic, the users and the features in two sentences?
- Is there a components diagram with an owner for every component, and a written specification for each?
- Does every boundary have an interface document that both sides have read, and does the code match it?
- Does the protocol document say, for every state, who speaks, what may be said, and what happens after an unexpected message? Is there a flowchart or state diagram?
- Did the owner of each component write tests for the neighboring component? Do the tests include stub-based, multithreaded and failure tests, and have we run at least once across several computers?
- Stop the server in the middle of a session: does every client say so clearly, keep the user's work, and recover when the server returns?
- Kill a client in the middle of a request: does the server carry on serving the others, and does it release that client's thread and socket?
- Does every blocking network call have a timeout?
- Is there a README that lets a newcomer build and run the system in ten minutes, plus the user, installation, performance and troubleshooting documents?

- Is the repository free of compiled files, passwords, tokens and real personal data? Does `main` compile and pass all tests?
- Does the history show every member's commits, with messages that explain why?
- Have we rehearsed the demonstration twice on the real machines, and do we have a backup recording?

15.12 Profitina in Practice

The order-status service is small, but its shape is Profitina's. Profitina is an AI visibility platform that measures how ready a brand's website is to be cited by AI answer engines (*generative engine optimization*, GEO). Its web application at app.profitina.com is a Python server built with FastAPI and served by Uvicorn, and every browser, phone and scheduled job using it is a client speaking HTTP and JSON (Section 15.8). The dashboard makes one request, `GET /api/me/dashboard`, and receives notifications, followed channels, rank and activity in one answer instead of a dozen requests: Equation (15.5) applied to a product.

The real-time community feed uses **Server-Sent Events** (SSE): the client keeps one HTTP connection open and the server pushes a line whenever something is posted, the `subscribe` idea of the sample brief, standardized for browsers. Production runs behind a Cloudflare Tunnel that provides TLS (Section 15.9), and a release counts as successful only when the new version answers `/api/health`, a health endpoint like the one in **Listing 15.23**. AI features run on a local model (Qwen2.5 through Ollama), so user data never leaves the server for AI processing.

Personal data is handled under UU PDP No. 27/2022 with **Table 15.6** turned into features: Argon2id password hashing, explicit consent, a login rate limit (Equation (15.6)), self-service data export and full account purge. Nothing is released until the automated suite of well over a thousand tests passes. And throughout, the application measures and advises; the user decides.

15.13 Common Mistakes, and How to Avoid Them

1. Forgetting to flush. The line stays in the buffer and both programs freeze. *Avoid it:* autoflush `PrintWriter` in Java; `flush()` after each message in Python.

2. Forgetting the newline. `print("Hello")` never completes the other side's `readLine`. *Avoid it:* `println`, or `"\n"` in Python; "one message, one line" in the protocol document.

3. Assuming one `recv` is one message. TCP is a byte stream. *Avoid it:* a framing rule and a reader that implements it (Section 15.4.4).

4. Serving one client at a time. *Avoid it:* hand each accepted socket to a pool thread (**Figure 15.6**).

5. Sharing unprotected state between client threads, which eventually loses updates. *Avoid it:* per-client objects, and `synchronized`, locks or atomics for the rest (Chapter 14).

6. Waiting forever on a silent client or an unreachable server. *Avoid it:* read timeouts on every connected socket and a connect timeout on every client.

7. "Address already in use". A second server on the same port fails, and on Linux and macOS a restarted Python server may fail even after the first has stopped:

```
EchoServer error: [Errno 98] Address already in use
EchoServer stopped
```

Avoid it: stop the old server or choose another port; set `SO_REUSEADDR` in Python servers on Linux and macOS; use port 0 in tests.

8. Mismatched encodings between a Windows client and a UTF-8 server. *Avoid it:* name UTF-8 on both sides of every stream.

9. Binding to all addresses while developing, exposing a half-finished server. *Avoid it:* `127.0.0.1` until you need more, then firewall and TLS.

10. Secrets in the source code or logs. *Avoid it:* environment variables, constant-time comparison, `***` in output.

11. Leaking personal data in errors and logs, such as "wrong phone number for Budi Santoso". *Avoid it:* identical, uninformative refusals; logs that say what happened, not whose data it was.

12. Coding before agreeing on the protocol. *Avoid it:* table, diagram and protocol object first, reviewed together.

15.14 Chapter Summary

- A **host** has an **IP address** (IPv4, 32 bits; IPv6, 128 bits); `127.0.0.1` is **loopback**, "this computer". A **port** (16 bits, Equation (15.1)) selects the program; port 0 asks the system for a free **ephemeral** port. A connection is identified by its **four-tuple** (Equation (15.2)).
- The Internet stacks four **layers** (link, internet, transport, application, **Figure 15.1**). **TCP** gives a reliable, ordered byte stream over a connection; **UDP** gives independent datagrams that keep their boundaries but may be lost (**Table 15.1**).
- A **server** binds and listens; a **client** connects. `accept()` returns a **new connected socket** per client while the **listening socket** stays open (**Figure 15.2**, **Figure 15.3**, **Figure 15.4**). Java uses `ServerSocket` and `Socket`; Python uses `socket.socket` with `bind`, `listen`, `accept` and `create_connection` (**Table 15.2**).
- Wrap socket streams in readers and writers with **UTF-8**, **flush** after each message, and use a **framing** rule; newline-delimited lines are the simplest. Every client follows five steps: open a socket, open streams, talk according to the protocol, close the streams, close the socket.
- **Design the protocol on paper first**: a table of states and messages, a state diagram, and a **protocol object** without I/O that can be tested alone (**Table 15.3**, **Figure 15.5**).
- Serve many clients with an **acceptor thread and a thread pool**; share as little as possible and protect what is shared (**Figure 15.6**, Equation (15.3)). Prove concurrency with a test that would fail without it.
- The **order-status service** speaks **JSON lines** (**Table 15.4**); a **proxy** object hides the wire format; because both implementations follow the same specification, the Java client works with the Python server and the Python client with the Java server. Round trips dominate cost (Equation (15.4), Equation (15.5)).
- **HTTP** is text over TCP with methods, paths, status codes and headers (**Table 15.5**); `HttpServer`/`HttpClient` and `http.server`/`urllib` speak it.
- **Security**: listen narrowly, validate everything, limit lengths, times and threads, keep secrets in the environment, encrypt with TLS, refuse without detail, rate-limit guessing (Equation (15.6)). **UU PDP No. 27/2022** asks for minimization, protection, limited retention and prompt breach notice (**Table 15.6**).
- **Test networked code** with in-process servers on port 0, a fresh server per test, and deliberate failures.

- **A team project** succeeds through a network-worthy topic, five components with owners and written **interfaces**, cross-testing, **Git** with feature branches and reviews, a **README**, deliberate handling of **server down** and **client down**, four kinds of documentation, and a rehearsed live demonstration.

15.15 Exercises

Exercise 15.1 (four-tuples). The counter laptop at `192.168.1.20` opens two connections to the order-status server at `192.168.1.10`, port 5052, and the staff tablet at `192.168.1.30` opens one. The operating systems assign the client ports 51234 and 51235 on the laptop and 49800 on the tablet. (a) Write the three four-tuples of Equation (15.2). (b) Which fields tell the server's operating system which connection a packet belongs to? (c) Using the ephemeral range of Section 15.2.2, at most how many connections can the laptop hold open to that one server port at the same time?

Exercise 15.2 (TCP or UDP). For each of the following, choose TCP or UDP and give a one-sentence reason with **Table 15.1**: (a) the order-status service; (b) each washing machine announcing "I am alive, 23 minutes left" every two seconds; (c) the live picture of the shop's security camera; (d) uploading the day's receipts as a PDF to the accountant; (e) looking up the address of `app.profitina.com`; (f) a multiplayer game sending every player's position thirty times a second.

Exercise 15.3 (making it freeze). Change `EchoClient` so that it sends with `out.print(userLine)` instead of `out.println(userLine)`. Predict what happens and why. Undo the change, and instead create the `PrintWriter` without autoflush. Predict again. Then add one line to the client that turns the freeze into an error message after three seconds, and state what the message says in Java and in Python.

Exercise 15.4 (extending a protocol on paper first). Add a command `help` to the knock-knock protocol: in any state, if the client says `help`, the server answers with a hint about what it expects next and stays in the same state. (a) Add the new rows to **Table 15.3**. (b) Say how **Figure 15.5** changes. (c) Change `processInput` in both languages. (d) Write three client lines that test the new rule.

Exercise 15.5 (a pool that is too small). In **Listing 15.13** and **Listing 15.14**, change the server's pool size from 3 to 2, keeping three clients. Before running it, predict what each client will print and how long the program will take. Then run it and explain the output with **Figure 15.6**.

Exercise 15.6 (the cost of round trips and threads). The laundry's server is in a data center with a one-way latency of 30 ms from the shop, over a 2 Mbit/s link. A status request is 100 bytes, a status response 200 bytes, and the server needs 1 ms per order. (a) With Equation (15.4) and Equation (15.5), how long does one status request take? (b) The owner's dashboard shows 25 orders. How long does it take to ask for them one by one? How long with one request that returns all 25 (response about 3000 bytes, server time 25 ms)? (c) A thread-per-client server with a stack size of 512 KB must hold 5000 idle clients. How much memory do the stacks reserve, by Equation (15.3)?

Exercise 15.7 (a client that retries). Write, in Java and in Python, a method `connectWithRetry(host, port, maxAttempts, afterFailure)` that tries to create a `LaundryStatusClient`, and after each refused connection waits 100 ms, then 200 ms, 400 ms and

so on, before trying again, giving up after `maxAttempts` tries. Demonstrate it with a server that is started only after the third failure. How long does a client wait in total before its k-th attempt?

Exercise 15.8 (a new command without personal data). Add a command `{"cmd":"queue"}` to the order-status service that answers how many orders are in each status, for the waiting-area screen, for example `{"ok":true,"RECEIVED":1,"WASHING":1,"DRYING":0,...}`. Change both servers, add one check to each test suite, and explain why this command needs neither a phone number nor a token.

Exercise 15.9 (HTTP status codes). If the order-status service were offered over HTTP, which status code of **Table 15.5** should each of these situations produce? (a) A valid status request for an existing order. (b) An order number that does not exist. (c) A body that is not JSON. (d) A staff command with no token. (e) A valid token that belongs to a courier, who may confirm deliveries but not move orders through washing. (f) The database behind the server is not reachable. (g) The sixth wrong phone guess for one order within a minute.

Exercise 15.10 (a security review). Review **Listing 15.17** as an attacker would. (a) Name four weaknesses that remain even though the server follows the seven habits of Section 15.9.1 as far as it goes. (b) Design a class `FailureLimiter` that allows five wrong phone guesses per order and then refuses that order for 60 seconds; say where the server calls it and how it must be protected against concurrent use. (c) Use Equation (15.6) to compare the expected guessing time without and with your limiter.

Exercise 15.11 (a chat room). Write, in Java and in Python, a chat server in which the first line a client sends is its name and every later line is broadcast to all connected clients as `name: text`, with `* name joined` and `* name left` announcements and a `/quit` command. Demonstrate it with three scripted members whose transcripts are printed at the end. Why must the broadcast hold a lock while it sends, and what goes wrong if one member's network is very slow?

Exercise 15.12 (a project brief of your own). Write the brief of Section 15.11.10 for a *Campus Lost-and-Found Board*: students post found items with a photo description and a place, owners claim them, and the security office confirms hand-overs. Include the pitch, at least five features, components with owners for a team of three, a protocol table with at least five requests, a four-week milestone table, performance criteria, the personal data involved and how you protect it, and five cross tests.

Exercise 15.13 (a merge conflict). Ayu changes the run line of the README to `java BoardServer 5060`, commits and pushes. At the same time, on his own branch, Bagus changes the same line to `java BoardServer 5050 --quiet` and commits. When Bagus merges `main` into his branch, what does Git print and what does the README look like? Resolve the conflict so that both intentions survive, and give the commands.

15.16 Solutions to the Exercises

Solution 15.1. (a) The three connections are (192.168.1.20, 51234, 192.168.1.10, 5052), (192.168.1.20, 51235, 192.168.1.10, 5052) and (192.168.1.30, 49800, 192.168.1.10, 5052). (b) All four fields are used together; here the server-side fields are the same for all three connections, so the client address separates the tablet from the laptop, and the client port separates the laptop's two connections. (c) The client address, the server address and the server port are fixed, so only p_c can vary. The ephemeral range 49152–65535 contains $65\,535 - 49\,152 + 1 = 16\,384$

ports, so at most 16,384 simultaneous connections, a limit that matters only for load-testing tools. (Linux uses the range 32768–60999 by default, which gives 28,232.)

Solution 15.2. (a) TCP: a lost or reordered status message would give a customer a wrong answer. (b) UDP: a lost heartbeat is harmless because another arrives two seconds later, and a missing series of heartbeats is itself the signal that a machine is down. (c) UDP: a late video frame is worthless, and resending it would only delay the next one. (d) TCP: every byte of the PDF must arrive, in order. (e) UDP: one small question and one small answer, where a lost query is simply asked again (DNS switches to TCP for very large answers). (f) UDP: the newest position replaces the old one thirty times a second, so resending an old one helps nobody.

Solution 15.3. With `print`, the line has no newline, so the server's `readLine` waits for the end of the line while the client's `readLine` waits for the echo: a deadlock across two programs. Without autoflush, the complete line never leaves the client's buffer, with the same result. Adding `socket.setSoTimeout(3000);` after opening the socket makes the blocked `readLine` throw `java.net.SocketTimeoutException` with the message `Read timed out`, printed as `I/O error: Read timed out`. In Python, `sock.settimeout(3)` makes `readline` raise `TimeoutError` (called `socket.timeout` before Python 3.10) with the message `timed out`. A timeout does not repair the bug; it turns an invisible freeze into a visible, explainable error.

Solution 15.4. (a) Three new rows, one per state that expects input:

Server state	Client says	Server answers	Next state
`SENT_KNOCK_KNOCK`	`help`	`Say "Who's there?"`	`SENT_KNOCK_KNOCK`
`SENT_CLUE`	`help`	`Say "<clue> who?"`	`SENT_CLUE`
`ANOTHER`	`help`	`Say y for another joke, n to stop`	`ANOTHER`

(b) Each of the three states gains a loop arrow labeled `"help" / hint` that returns to the same state; no state is added. (c) The cleanest change checks `help` once, before the `switch`, so that the rule is written once:

```
Java, at the top of processInput:
    if (input != null && input.equalsIgnoreCase("help") && state != State.WAITING) {
        switch (state) {
            case SENT_KNOCK_KNOCK: return "Say \"Who's there?\"";
            case SENT_CLUE:        return "Say \"" + CLUES[joke] + " who?\"";
            default:               return "Say y for another joke, n to stop";
        }
    }

Python, at the top of process_input:
    if text is not None and text.lower() == "help" and self._state is not State.WAITING:
        if self._state is State.SENT_KNOCK_KNOCK:
            return "Say \"Who's there?\""
        if self._state is State.SENT_CLUE:
            return f"Say \"{CLUES[self._joke]} who?\""
        return "Say y for another joke, n to stop"
```

(d) The script `help`, `Who's there?`, `help`, `Lettuce who?`, `help`, `n` must produce, after `Knock! Knock!`: `Say "Who's there?"`, `Lettuce`, `Say "Lettuce who?"`, the answer with `Want another? (y/n)`, `Say y for another joke, n to stop`, `Bye.`. Note that before this change, `help` in state `ANOTHER` meant "stop": a protocol change can change the meaning of old messages, which is why both sides must agree on it on paper.

Solution 15.5. Prediction: the two workers greet two clients, which then wait at the barrier for a third client who is never greeted, because its connection sits in the pool's queue with no free worker. After five seconds the barrier times out; one waiting client gets a timeout, the other a

broken barrier, and the third client's own five-second read timeout has expired too. The program takes about five seconds and reports `false`. One Java run printed:

```
server started on an ephemeral port
--- client A ---
failed: TimeoutException
--- client B ---
failed: SocketTimeoutException
--- client C ---
failed: BrokenBarrierException
server stopped; clients served: 3
all clients were greeted before any finished: false
```

and the Python version:

```
server started on an ephemeral port
--- client A ---
failed: BrokenBarrierError
--- client B ---
failed: BrokenBarrierError
--- client C ---
failed: BrokenBarrierError
server stopped; clients served: 3
all clients were greeted before any finished: false
```

Which client gets which exception varies from run to run; the verdict `false` does not. In Python, a timeout at a `threading.Barrier` breaks it for everyone, so all three report `BrokenBarrierError`. A pool smaller than the number of conversations that must be active *simultaneously* becomes a queue, and a queue behind a conversation that waits for another member of the queue is a deadlock: Chapter 14's dining philosophers in a new costume.

Solution 15.6. (a) By Equation (15.5), $\text{RTT} = 2 \times 30\ \text{ms} + \frac{(100+200)\times 8\ \text{bits}}{2\times 10^6\ \text{bits/s}} = 60\ \text{ms} + 1.2\ \text{ms} = 61.2\ \text{ms}$, and one request takes $61.2 + 1 = 62.2$ ms. (b) One by one: $25 \times 62.2 = 1555$ ms, about 1.6 seconds, long enough for the owner to notice. One combined request: $\text{RTT} = 60 + \frac{(100+3000)\times 8}{2\times 10^6} \times 1000 = 60 + 12.4 = 72.4$ ms, plus 25 ms of server time, about 97 ms: sixteen times faster, although it carries more bytes. (c) By Equation (15.3), $M \approx 5000 \times 512\ \text{KB} = 2\,560\,000$ KB, about 2.4 GiB of stack reserved by threads that are doing nothing, which is why such a server needs a pool, virtual threads or `asyncio`.

Solution 15.7. **Listing 15.27** and **Listing 15.28** are the two programs. `main` first obtains a free port and closes it, so nobody listens there, and the callback starts the server on that port exactly after the third failure; no timing tricks are needed.

Listing 15.27 `RetryDemo.java`: connecting with exponential backoff; the server appears after the third failure.

```
import java.io.IOException;
import java.net.ConnectException;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.util.function.IntConsumer;

/** A client that does not give up at the first "connection refused". */
public class RetryDemo {
    /** Tries up to maxAttempts times, waiting 100, 200, 400, ... ms between the tries. */
    static LaundryStatusClient connectWithRetry(String host, int port, int maxAttempts,
            IntConsumer afterFailure) throws IOException, InterruptedException {
        long delayMs = 100;
        for (int attempt = 1; ; attempt++) {
            try {
                LaundryStatusClient client = new LaundryStatusClient(host, port);
                System.out.println("attempt " + attempt + ": connected");
                return client;
            } catch (ConnectException e) {
                System.out.println("attempt " + attempt + ": server not reachable");
                if (attempt == maxAttempts) {
                    throw e;
                }
                afterFailure.accept(attempt);
                System.out.println("  waiting " + delayMs + " ms");
                Thread.sleep(delayMs);
                delayMs *= 2;  // exponential backoff: do not hammer a server that is down
            }
        }
    }

    public static void main(String[] args) throws Exception {
        int port;
        try (ServerSocket probe = new ServerSocket(0, 1, InetAddress.getLoopbackAddress()))
{
            port = probe.getLocalPort();  // a free port; nobody listens there after this
        }
        LaundryStatusServer server = LaundryStatusServer.withSampleOrders(null, false);
```

Listing 15.27 (Continued) *RetryDemo.java: connecting with exponential backoff; the server appears after the third failure.*

```
        IntConsumer startServerAfterThirdFailure = attempt -> {
            if (attempt == 3) {
                try {
                    server.start(port);
                    System.out.println("  (the server comes up now)");
                } catch (IOException e) {
                    System.out.println("  (the server could not start: " + e.getMessage() +
")");
                }
            }
        };
        try (LaundryStatusClient client = connectWithRetry("127.0.0.1", port, 5,
                startServerAfterThirdFailure)) {
            System.out.println("ping -> " + client.ping().get("reply"));
        }
        server.stop();
        System.out.println("server stopped");

        try {
            connectWithRetry("127.0.0.1", port, 2, attempt -> { });
        } catch (ConnectException e) {
            System.out.println("gave up after 2 attempts; tell the user the service is
down");
        }
    }
}
```

```
attempt 1: server not reachable
  waiting 100 ms
attempt 2: server not reachable
  waiting 200 ms
attempt 3: server not reachable
  (the server comes up now)
  waiting 400 ms
attempt 4: connected
ping -> pong
server stopped
attempt 1: server not reachable
  waiting 100 ms
attempt 2: server not reachable
gave up after 2 attempts; tell the user the service is down
```

Listing 15.28 `retry_demo.py`: the same retrying client in Python.

```
"""retry_demo.py: a client that does not give up at the first "connection refused"."""
import socket
import time

from laundry_status_client import LaundryStatusClient
from laundry_status_server import LaundryStatusServer, OrderBook


def connect_with_retry(host, port, max_attempts, after_failure):
    """Try up to max_attempts times, waiting 100, 200, 400, ... ms between the tries."""
    delay_ms = 100
    attempt = 1
    while True:
        try:
            client = LaundryStatusClient(host, port)
            print(f"attempt {attempt}: connected")
            return client
        except ConnectionRefusedError:
            print(f"attempt {attempt}: server not reachable")
            if attempt == max_attempts:
                raise
            after_failure(attempt)
            print(f"  waiting {delay_ms} ms")
            time.sleep(delay_ms / 1000)
            delay_ms *= 2  # exponential backoff: do not hammer a server that is down
            attempt += 1


def main():
    with socket.socket() as probe:
        probe.bind(("127.0.0.1", 0))
        port = probe.getsockname()[1]  # a free port; nobody listens there after this
    server = LaundryStatusServer(OrderBook.sample(), None, verbose=False)

    def start_server_after_third_failure(attempt):
        if attempt == 3:
            server.start(port)
            print("  (the server comes up now)")

    with connect_with_retry("127.0.0.1", port, 5, start_server_after_third_failure) as
client:
        print("ping -> " + client.ping()["reply"])
    server.stop()
    print("server stopped")
```

Listing 15.28 (Continued) *retry_demo.py: the same retrying client in Python.*

```
    try:
        connect_with_retry("127.0.0.1", port, 2, lambda attempt: None)
    except ConnectionRefusedError:
        print("gave up after 2 attempts; tell the user the service is down")


if __name__ == "__main__":
    main()
```

```
attempt 1: server not reachable
  waiting 100 ms
attempt 2: server not reachable
  waiting 200 ms
attempt 3: server not reachable
  (the server comes up now)
  waiting 400 ms
attempt 4: connected
ping -> pong
server stopped
attempt 1: server not reachable
  waiting 100 ms
attempt 2: server not reachable
gave up after 2 attempts; tell the user the service is down
```

Before the k-th attempt the client has waited $100 + 200 + \cdots + 100 \cdot 2^{k-2} = 100\,(2^{k-1} - 1)$ ms: 700 ms before the fourth attempt, 1.5 s before the fifth. This *exponential backoff* keeps a crowd of clients from hammering a restarting server; production clients also add a random *jitter* and cap the wait. And after `maxAttempts`, the client gives up and says so.

Solution 15.8. Add to `OrderBook` a method that counts under the same lock as every other method, then one branch in `handle`:

```
Java (OrderBook):
    synchronized Map<String, Object> countByStatus() {
        Map<String, Object> counts = Json.object("ok", true);
        for (Status s : Status.values()) {
            counts.put(s.name(), 0L);
        }
        for (LaundryOrder o : orders.values()) {
            counts.put(o.status.name(), (Long) counts.get(o.status.name()) + 1);
        }
        return counts;
    }
Java (handle):   } else if ("queue".equals(cmd)) { return book.countByStatus(); }

Python (OrderBook):
    def count_by_status(self):
        with self._lock:
            counts = {"ok": True, **{s.name: 0 for s in Status}}
            for o in self._orders.values():
                counts[o.status.name] += 1
            return counts
Python (handle):  if cmd == "queue": return self._book.count_by_status()

Check / test:
    expect(1L, c.send(Json.object("cmd", "queue")).get("WASHING"));          // Java
    self.assertEqual(1, c.send({"cmd": "queue"})["WASHING"])                  # Python
```

With the sample orders both servers answer `{"ok":true,"RECEIVED":1,"WASHING":1,"DRYING":0,"IRONING":0,"READY":1,"PICKED_UP":0}`. Every status starts at zero so the screen always receives the same keys, and Java uses `Long` because that is what `Json.parse` produces on the client side. The command needs no phone digits and no token because its answer is aggregated and reveals nothing about any person or single order: data minimization (**Table 15.6**) as a design rule.

Solution 15.9. (a) 200 OK. (b) 404 Not Found; following habit 6 of Section 15.9.1, a wrong phone number should get the same 404 and body. (c) 400 Bad Request. (d) 401 Unauthorized: credentials missing. (e) 403 Forbidden: valid credentials, insufficient rights. (f) 503 Service Unavailable, a temporary fault outside the request, telling clients to retry later (500 is also defensible). (g) 429 Too Many Requests, ideally with a `Retry-After` header.

Solution 15.10. (a) *No encryption*: anyone on the network can read order numbers, phone digits and the staff token (fix: TLS). *No rate limit*: four digits fall in seconds by Equation (15.6). *One shared staff token*: the log cannot say who moved an order, and a departing employee keeps the secret (fix: personal, revocable tokens). *Predictable order numbers*: a guesser walks from ORD-0001 upward (fix: random codes, or the rate limit). (b) A sketch:

```
class FailureLimiter {                        // one shared object, used by every worker
    private final Map<String, Integer> failures = new HashMap<>();
    private final Map<String, Long> lockedUntil = new HashMap<>();

    synchronized boolean isLocked(String order, long nowMs) {
        Long until = lockedUntil.get(order);
        return until != null && nowMs < until;
    }

    synchronized void recordFailure(String order, long nowMs) {
        int n = failures.merge(order, 1, Integer::sum);
        if (n >= 5) {
            lockedUntil.put(order, nowMs + 60_000);
            failures.remove(order);
        }
    }

    synchronized void recordSuccess(String order) {
        failures.remove(order);
    }
}
```

In `handle`, the `status` branch first calls `isLocked(order, now)` and, if locked, refuses with the usual message (over HTTP, 429); otherwise it calls `publicView`, then `recordFailure` or `recordSuccess`. Passing the time in makes the class testable without waiting a minute. The methods are `synchronized` because workers may record failures for one order at once, and two simultaneous failures must not count as one, the lost update of Chapter 14. A production version would also forget old entries. (c) Without a limit, at $r = 1000$ per second, Equation (15.6) gives about 5 seconds; with it, $r = 1/12$ per second gives $5000.5 \times 12 \approx 60\,000$ seconds, almost 17 hours, with the logs filling with refusals staff can notice.

Solution 15.11. **Listing 15.29** and **Listing 15.30** give every member a thread, keep the members' writers in a map, and broadcast every line to all. The demonstration's scripted members record what they receive, and each step waits until every member has received what it should, so the output is the same on every run.

Listing 15.29 `ChatRoom.java`: a chat room that broadcasts every line, with scripted members for a repeatable demonstration.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

/** A chat room: every line one member sends is broadcast to every member. */
public class ChatRoom {
    private final Map<String, PrintWriter> members = new LinkedHashMap<>();  // guarded by
this
    private final Set<Socket> sockets = ConcurrentHashMap.newKeySet();
    private ServerSocket serverSocket;

    public int start(int port) throws IOException {
        serverSocket = new ServerSocket(port, 50, InetAddress.getLoopbackAddress());
        Thread acceptor = new Thread(() -> {
            while (true) {
                try {
                    Socket socket = serverSocket.accept();
                    new Thread(() -> serve(socket)).start();  // one thread per member
                } catch (IOException e) {
                    return;  // stop() closed the server socket
                }
            }
        });
        acceptor.start();
        return serverSocket.getLocalPort();
    }
```

Listing 15.29 (Continued) *ChatRoom.java: a chat room that broadcasts every line, with scripted members for a repeatable demonstration.*

```
    private void serve(Socket socket) {
        sockets.add(socket);
        String name = null;
        try (socket;
             BufferedReader in = new BufferedReader(new InputStreamReader(
                     socket.getInputStream(), StandardCharsets.UTF_8));
             PrintWriter out = new PrintWriter(new OutputStreamWriter(
                     socket.getOutputStream(), StandardCharsets.UTF_8), true)) {
            String wanted = in.readLine();  // protocol: the first line is the member's
name
            if (wanted == null || !join(wanted, out)) {
                out.println("! name missing or taken");
                return;
            }
            name = wanted;
            broadcast("* " + name + " joined");
            String line;
            while ((line = in.readLine()) != null && !line.equals("/quit")) {
                broadcast(name + ": " + line);
            }
        } catch (IOException e) {
            // the member vanished; treat it exactly like /quit
        } finally {
            sockets.remove(socket);
            if (name != null) {
                leave(name);
                broadcast("* " + name + " left");
            }
        }
    }

    private synchronized boolean join(String name, PrintWriter out) {
        if (name.isBlank() || members.containsKey(name)) {
            return false;
        }
        members.put(name, out);
        return true;
    }

    private synchronized void leave(String name) {
        members.remove(name);
    }
```

Listing 15.29 (Continued) *ChatRoom.java: a chat room that broadcasts every line, with scripted members for a repeatable demonstration.*

```
    /** Holding the lock while sending means every member sees the lines in the same order.
*/
    private synchronized void broadcast(String line) {
        for (PrintWriter out : members.values()) {
            out.println(line);
        }
    }

    public void stop() throws IOException {
        serverSocket.close();
        for (Socket socket : sockets) {
            socket.close();
        }
    }

    /** A scripted member for the demonstration: it records every line it receives. */
    static class Member implements AutoCloseable {
        final String name;
        private final Socket socket;
        private final PrintWriter out;
        private final List<String> received = new ArrayList<>();

        Member(String name, int port) throws IOException {
            this.name = name;
            socket = new Socket(InetAddress.getLoopbackAddress(), port);
            out = new PrintWriter(new OutputStreamWriter(
                    socket.getOutputStream(), StandardCharsets.UTF_8), true);
            BufferedReader in = new BufferedReader(new InputStreamReader(
                    socket.getInputStream(), StandardCharsets.UTF_8));
            Thread listener = new Thread(() -> {
                try {
                    String line;
                    while ((line = in.readLine()) != null) {
                        synchronized (this) {
                            received.add(line);
                            notifyAll();
                        }
                    }
                } catch (IOException e) {
                    // socket closed: nothing more to hear
                }
            });
            listener.setDaemon(true);
            listener.start();
            say(name);
        }
```

Listing 15.29 (Continued) *ChatRoom.java: a chat room that broadcasts every line, with scripted members for a repeatable demonstration.*

```
        void say(String line) {
            out.println(line);
        }

        /** Waits (at most 5 s) until this member has received n lines in total. */
        synchronized void awaitLines(int n) throws InterruptedException {
            long deadline = System.currentTimeMillis() + 5000;
            while (received.size() < n && System.currentTimeMillis() < deadline) {
                wait(100);
            }
        }

        synchronized List<String> received() {
            return new ArrayList<>(received);
        }

        @Override
        public void close() throws IOException {
            socket.close();
        }
    }

    public static void main(String[] args) throws Exception {
        ChatRoom room = new ChatRoom();
        int port = room.start(0);
        Member ayu = new Member("Ayu", port);
        ayu.awaitLines(1);
        Member budi = new Member("Budi", port);
        ayu.awaitLines(2);
        budi.awaitLines(1);
        ayu.say("Is machine W2 free?");
        budi.awaitLines(2);
        budi.say("Yes, go ahead.");
        ayu.awaitLines(4);
        Member citra = new Member("Citra", port);
        ayu.awaitLines(5);
        citra.say("Hi all!");
        budi.awaitLines(5);
        budi.say("/quit");
        citra.awaitLines(3);
        ayu.say("Bye, Budi!");
        ayu.awaitLines(8);
        citra.awaitLines(4);
```

Listing 15.29 (Continued) *ChatRoom.java: a chat room that broadcasts every line, with scripted members for a repeatable demonstration.*

```
        for (Member m : List.of(ayu, budi, citra)) {
            System.out.println("--- what " + m.name + " saw ---");
            m.received().forEach(System.out::println);
        }
        for (Member m : List.of(ayu, budi, citra)) {
            m.close();  // only now: closing earlier would add "* Ayu left" to Citra's view
        }
        room.stop();
    }
}
```

```
--- what Ayu saw ---
* Ayu joined
* Budi joined
Ayu: Is machine W2 free?
Budi: Yes, go ahead.
* Citra joined
Citra: Hi all!
* Budi left
Ayu: Bye, Budi!
--- what Budi saw ---
* Budi joined
Ayu: Is machine W2 free?
Budi: Yes, go ahead.
* Citra joined
Citra: Hi all!
--- what Citra saw ---
* Citra joined
Citra: Hi all!
* Budi left
Ayu: Bye, Budi!
```

Listing 15.30 `chat_room.py`: the chat room in Python.

```
"""chat_room.py: every line one member sends is broadcast to every member."""
import socket
import sys
import threading
import time


class ChatRoom:
    def __init__(self):
        self._members = {}  # name -> writer, guarded by self._lock
        self._lock = threading.Lock()
        self._sockets = set()
        self._server_socket = None

    def start(self, port):
        self._server_socket = socket.socket()
        if sys.platform != "win32":  # reuse the port at once after a restart
            self._server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self._server_socket.bind(("127.0.0.1", port))
        self._server_socket.listen(50)
        threading.Thread(target=self._accept_loop, daemon=True).start()
        return self._server_socket.getsockname()[1]

    def _accept_loop(self):
        while True:
            try:
                conn, _ = self._server_socket.accept()
            except OSError:
                return  # stop() closed the server socket
            # one thread per member
            threading.Thread(target=self._serve, args=(conn,), daemon=True).start()

    def _serve(self, conn):
        self._sockets.add(conn)
        name = None
        try:
            with conn, conn.makefile("r", encoding="utf-8", newline="\n") as reader, \
                    conn.makefile("w", encoding="utf-8", newline="\n") as writer:
                wanted = reader.readline().rstrip("\n")  # protocol: the first line is the
name
                if not self._join(wanted, writer):
                    writer.write("! name missing or taken\n")
                    writer.flush()
                    return
                name = wanted
                self._broadcast(f"* {name} joined")
                for line in reader:
```

Listing 15.30 (Continued) *chat_room.py: the chat room in Python.*

```
                    line = line.rstrip("\n")
                    if line == "/quit":
                        break
                    self._broadcast(f"{name}: {line}")
        except OSError:
            pass  # the member vanished; treat it exactly like /quit
        finally:
            self._sockets.discard(conn)
            if name is not None:
                self._leave(name)
                self._broadcast(f"* {name} left")

    def _join(self, name, writer):
        with self._lock:
            if not name.strip() or name in self._members:
                return False
            self._members[name] = writer
            return True

    def _leave(self, name):
        with self._lock:
            self._members.pop(name, None)

    def _broadcast(self, line):
        """Holding the lock while sending means every member sees the lines in the same
order."""
        with self._lock:
            for writer in self._members.values():
                try:
                    writer.write(line + "\n")
                    writer.flush()
                except OSError:
                    pass  # that member is leaving; its own thread cleans up

    def stop(self):
        self._server_socket.close()
        for conn in list(self._sockets):
            try:
                conn.shutdown(socket.SHUT_RDWR)
            except OSError:
                pass


class Member:
    """A scripted member for the demonstration: it records every line it receives."""
```

Listing 15.30 (Continued) *chat_room.py: the chat room in Python.*

```
    def __init__(self, name, port):
        self.name = name
        self._sock = socket.create_connection(("127.0.0.1", port))
        self._writer = self._sock.makefile("w", encoding="utf-8", newline="\n")
        self._reader = self._sock.makefile("r", encoding="utf-8", newline="\n")
        self._received = []
        self._cond = threading.Condition()
        threading.Thread(target=self._listen, daemon=True).start()
        self.say(name)

    def _listen(self):
        try:
            for line in self._reader:
                with self._cond:
                    self._received.append(line.rstrip("\n"))
                    self._cond.notify_all()
        except (OSError, ValueError):
            pass  # socket closed: nothing more to hear

    def say(self, line):
        self._writer.write(line + "\n")
        self._writer.flush()

    def await_lines(self, n):
        """Wait (at most 5 s) until this member has received n lines in total."""
        deadline = time.monotonic() + 5
        with self._cond:
            while len(self._received) < n and time.monotonic() < deadline:
                self._cond.wait(0.1)

    def received(self):
        with self._cond:
            return list(self._received)

    def close(self):
        self._sock.close()
```

Listing 15.30 (Continued) *chat_room.py: the chat room in Python.*

```
def main():
    room = ChatRoom()
    port = room.start(0)
    ayu = Member("Ayu", port)
    ayu.await_lines(1)
    budi = Member("Budi", port)
    ayu.await_lines(2)
    budi.await_lines(1)
    ayu.say("Is machine W2 free?")
    budi.await_lines(2)
    budi.say("Yes, go ahead.")
    ayu.await_lines(4)
    citra = Member("Citra", port)
    ayu.await_lines(5)
    citra.say("Hi all!")
    budi.await_lines(5)
    budi.say("/quit")
    citra.await_lines(3)
    ayu.say("Bye, Budi!")
    ayu.await_lines(8)
    citra.await_lines(4)

    for m in (ayu, budi, citra):
        print(f"--- what {m.name} saw ---")
        for line in m.received():
            print(line)
    for m in (ayu, budi, citra):
        m.close()  # only now: closing earlier would add "* Ayu left" to Citra's view
    room.stop()


if __name__ == "__main__":
    main()
```

```
--- what Ayu saw ---
* Ayu joined
* Budi joined
Ayu: Is machine W2 free?
Budi: Yes, go ahead.
* Citra joined
Citra: Hi all!
* Budi left
Ayu: Bye, Budi!
--- what Budi saw ---
* Budi joined
Ayu: Is machine W2 free?
Budi: Yes, go ahead.
* Citra joined
Citra: Hi all!
--- what Citra saw ---
* Citra joined
Citra: Hi all!
* Budi left
Ayu: Bye, Budi!
```

Each member sees only what was said while it was present. `broadcast` holds the room's lock for two reasons. The member map must not change while it is walked (otherwise `ConcurrentModificationException` in Java, "dictionary changed size during iteration" in Python). And when two members speak at once, one broadcast finishes before the other starts, so everyone sees the same order. The price is the slow-member problem: writing to a member whose socket buffer is full blocks while holding the lock, so *everyone* waits for the slowest. Real

chat servers give each member an outgoing queue and writer thread (Chapter 14's producer–consumer) and disconnect members whose queue grows too long.

Solution 15.12. A worked brief:

Title and pitch. Campus Lost-and-Found Board. Students lose cards, flash drives and umbrellas every week, and the security office keeps a paper list nobody reads. Finders post items, every screen shows them at once, owners claim items by describing them, and the security office confirms hand-overs.

Features. F1 post a found item (category, description, place, time); F2 live list of unclaimed items, filtered by category; F3 claim with a private detail only the owner knows; F4 the security office compares and confirms or rejects; F5 items unclaimed after 60 days marked for donation; F6 data survives restarts (SQLite).

Components and owners. Member 1: server and database behind an `ItemRepository` interface. Member 2: client library, protocol, reconnection. Member 3: GUI for students and security office, testing lead.

Protocol (JSON lines).

Request	Response	Refusals
`{"cmd":"login","nim":…,"pin":…}`	`{"ok":true,"role":"student"}` or `"security"`	wrong credentials (same message for both fields); too many attempts
`{"cmd":"post","category":…,"description":…,"place":…}`	`{"ok":true,"item":"F-1043"}`	missing field; description over 300 characters
`{"cmd":"list","category":…}`	`{"ok":true,"items":[…]}` (no finder identity)	unknown category
`{"cmd":"claim","item":…,"detail":…}`	`{"ok":true,"claim":"C-77"}`	item already handed over
`{"cmd":"decide","claim":…,"accept":true}`	`{"ok":true}`	not the security role
`{"cmd":"subscribe"}`	then pushed `{"event":"posted",…}` / `{"event":"closed",…}`	—
`{"cmd":"bye"}`	`{"ok":true}`, then close	—

Milestones. Week 1: specification, protocol, session state diagram, components diagram, repository. Week 2: server, client library and GUI each against stubs. Week 3: integration, F1–F4, 30-client load test. Week 4: F5–F6, failure demonstrations on three computers, documentation, rehearsals.

Performance criteria. 30 simultaneous clients; a new post visible on all screens within 2 seconds on campus Wi-Fi; list response within 300 ms for 500 items.

Personal data. Student numbers and PINs (stored only as salted hashes, never logged); finders never shown to other students; claim details visible only to security; closed items deleted after 90 days; invented test data.

Cross tests. (1) Member 2: the server refuses `decide` from a student. (2) Member 2: `list` never contains a finder's student number. (3) Member 1: the client reconnects and re-subscribes after a server restart. (4) Member 1: the client reports, not throws, a refusal for a 301-character description. (5) Member 3: with 30 scripted clients, a post reaches all within 2 seconds.

Solution 15.13. When Bagus runs `git merge main`, Git prints

```
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

and the README now contains both versions between markers:

```
<<<<<<< HEAD
    cd server && javac *.java && java BoardServer 5050 --quiet
=======
    cd server && javac *.java && java BoardServer 5060
>>>>>>> main
```

`HEAD` is Bagus's branch and `main` what he merged in. Both intentions fit in one line, `cd server && javac *.java && java BoardServer 5060 --quiet`. Bagus replaces the five lines by it, removes the markers, checks with `git diff`, then `git add README.md` and `git commit`, runs the program and tests before `git push`, and tells Ayu, because a conflict means two people were deciding the same thing without talking.

References

[1] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 8th ed. Hoboken, NJ, USA: Pearson, 2021.

[2] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 8200, IETF, Jul. 2017.

[3] M. Cotton, L. Eggert, J. Touch, M. Westerlund, and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry," RFC 6335, IETF, Aug. 2011.

[4] W. Eddy, Ed., "Transmission Control Protocol (TCP)," RFC 9293, IETF, Aug. 2022.

[5] J. Postel, "User Datagram Protocol," RFC 768, IETF, Aug. 1980.

[6] F. Yergeau, "UTF-8, a transformation format of ISO 10646," RFC 3629, IETF, Nov. 2003.

[7] Python Software Foundation, "socket — Low-level networking interface," and G. McMillan, "Socket Programming HOWTO," *The Python Standard Library*, version 3.14. [Online]. Available: https://docs.python.org/3/library/socket.html

[8] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming, Volume 1: The Sockets Networking API*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2004.

[9] Oracle, "Lesson: All About Sockets," *The Java Tutorials*, Custom Networking trail. [Online]. Available: https://docs.oracle.com/javase/tutorial/networking/sockets/

[10] R. Pressler and A. Bateman, "JEP 444: Virtual Threads," OpenJDK, 2023. [Online]. Available: https://openjdk.org/jeps/444

[11] T. Bray, Ed., "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 8259, IETF, Dec. 2017.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

[13] R. Fielding, M. Nottingham, and J. Reschke, Eds., "HTTP Semantics," RFC 9110, IETF, Jun. 2022.

[14] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi* (Law No. 27 of 2022 on Personal Data Protection), Lembaran Negara Tahun 2022 Nomor 196, Oct. 2022.

[15] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, IETF, Aug. 2018.

[16] S. Chacon and B. Straub, *Pro Git*, 2nd ed. New York, NY, USA: Apress, 2014. [Online]. Available: https://git-scm.com/book

Chapter 16

Evaluation and Discussion IV: Iterators, Concurrency, Networking, and the Whole Book

This is the last chapter of the book, and like Chapters 4, 8 and 12 it adds no new language features. It works problems instead, and this time the problems are big. The first eight come from one team project brief: build a client–server application with a graphical user interface that talks over sockets. We solve that brief the way a small team really would, one deliverable at a time, on a running example that is small enough to print in full: a knock-knock joke server with a console client and a window client. Along the way we decide who does what, write the interfaces between the members, turn a flow chart into a state machine, serve many clients at once, test each side of the network against a stub of the other, make both sides notice when the other disappears, keep a window responsive while the network waits, and measure the numbers that the documentation promises. Six more problems are new and each isolates one idea from Chapters 13 to 15: a collection with its own iterator, an immutable value object that is safe as a map key, a shared counter with a race we measure and then remove, a producer–consumer wash line, a tiny request–reply protocol of our own, and finally a capstone that combines everything in the book, a small but complete Profitina Laundry system with classes, interfaces, inheritance, collections, exceptions, files, the Model–View–Controller pattern and a socket API, written twice, in Java and in Python, with its tests. Every problem follows the route you know from the earlier evaluation chapters: the statement, the problem in our own words, a design, the Java solution, the Python solution, the real output, a line-by-line explanation, a tempting wrong solution and exactly how it fails, and a *check yourself* variant with its answer. The chapter closes with a map of the whole book, a guide to what to learn next, and a closing word.

16.1 Where We Left Off

Chapters 13, 14 and 15 took the object-oriented toolbox of the first twelve chapters out of the single-threaded, single-machine world. It is worth naming, before we start, exactly which tools they added, because naming the tool is half of choosing it. **Table 16.1** lists them.

Table 16.1 The tools added by Chapters 13 to 15, in both languages. Every problem of this chapter uses several of them.

Chapter	Tool	In Java	In Python
13	walking through a collection	`Iterator`, `Iterable`, the for-each loop	`__iter__`, `__next__`, generators with `yield`
13	changing a collection while walking	`ConcurrentModificationException`, `Iterator.remove`, `removeIf`	`RuntimeError` for dicts and sets, comprehensions
13	ordering objects	`Comparable`, `Comparator`	`key=` functions, `__lt__`
13	copying	shallow and deep copies	`copy.copy`, `copy.deepcopy`
13	immutability	`final` fields, records, unmodifiable views	frozen dataclasses, tuples
14	threads	`Thread`, `Runnable`, `start`, `join`, `interrupt`	`threading.Thread`, `join`
14	races and their cure	`synchronized`, locks, atomic variables	`threading.Lock`, `with lock:`
14	waiting for a condition	`wait`, `notifyAll`, `BlockingQueue`	`threading.Condition`, `queue.Queue`
14	deadlock and livelock	lock ordering, timeouts	the same ideas
15	sockets	`ServerSocket`, `Socket`, streams	`socket.create_server`, `create_connection`, `makefile`
15	protocols	the knock-knock protocol, one thread per client	the same

The rest of the book is still here, of course, and the capstone problem at the end of this chapter uses almost all of it: classes and objects (Chapter 3), arrays, lists, exceptions and files (Chapter 5), documentation, tests and interfaces (Chapter 6), inheritance and polymorphism (Chapter 7), collections and generics (Chapter 9), graphical interfaces and events (Chapter 10) and the Model–View–Controller pattern (Chapter 11).

The chapter is organized in five layers.

1. **A method** (Section 16.2): ten steps that take a small team from a one-page brief to a tested, documented client–server application.
2. **The brief, restated** (Section 16.3), with a table that maps each of its demands to one worked problem.
3. **Eight problems that carry out the brief** (Sections 16.4 to 16.6), Problems 16.1 to 16.8, all on the knock-knock example.
4. **Six new problems** (Sections 16.7 to 16.10), Problems 16.9 to 16.14, ending with the capstone.
5. **The closing sections**: a comparison of all fourteen problems, a map of the whole book, where to go next, Profitina in practice, common mistakes, the summary and a closing word, exercises and their full solutions.

Each problem is laid out in the same order, with bold labels you can find again: **The problem**, **In our own words**, **Design**, **The Java solution**, **The Python solution**, **Line by line**, **A tempting wrong solution**, and **Check yourself**. As in Chapter 8, the Java programs test themselves with a small `check` method that prints `PASS` and a name, and the Python programs use `unittest` [1]. The output printed after each listing is therefore also its test report.

> **HOW TO USE THIS CHAPTER.** The problems in the first half build on each other: Problem 16.3 uses the classes of Problems 16.1 and 16.2, and so on. Read them in order the first time. Before each solution, close the book for ten minutes and sketch your own design on paper: the classes, the messages, the test you would write first. Then compare. Programs about threads and networks are the ones where "I read it and it made sense" is furthest from "I could write it", so type at least the smaller programs yourself and run them.

A note about output. Most programs of this chapter print exactly the same lines on every run, because every test was written so that it does not depend on luck. A few programs *measure* something, a time or a number of lost updates, and those numbers change from run to run and from machine to machine. The text says so every time, and the output shown is the real output of one run on the machine that built this book.

16.2 How to Approach a Whole-System Project

Until now, every problem in this book fitted in one head and one file. A client–server application with a window does not. It has at least two programs that run at the same time, often on different machines; it has several threads inside each program; it has a protocol that both programs must follow exactly; and it is usually built by several people at once. The main difficulty is no longer "how do I write this method?" but "how do we make sure all the parts fit together when they are finally joined?" This section gives a method of ten steps. They are the eight steps of Section 8.2 grown up for a team and a network.

> **DEFINITION.** A *contract* is a written promise at a boundary between two parts of a system: what one side may ask, what the other side promises to answer, and what happens when something goes wrong. A Java `interface` is one kind of contract; a network protocol is another; a file format is a third.

Step 1 — Choose a topic that is small in the middle. A good project has a small core and room to grow at the edges. "A chat with a game" is fine if the first version of the game is tic-tac-toe. Write one sentence that says what a user can do with the finished program, and one sentence that says what it will *not* do.

Step 2 — Write the specification as things a user does. "A user connects, sees the list of rooms, joins one, sends messages, sees messages from others, and leaves." Each sentence becomes a feature and, later, a test.

Step 3 — Draw the parts and give each one an owner. A client–server application has five natural parts: the server, the client logic, the graphical interface, the storage (a database or files), and the testing and documentation that hold everything together. With three people, one person owns two parts. Ownership means "I make it work and I answer questions about it", not "nobody else may touch it".

Step 4 — Write the contracts before the code. For every arrow between two parts write the contract: a Java interface or a Python abstract class for parts inside one program, a protocol for parts on two sides of the network. Put the contracts in the repository on the first day, even if nothing implements them yet.

Step 5 — Design the protocol as a flow chart, then as a state machine. A flow chart shows the conversation for people; a state machine, with a table of "in this state, on this input, reply this and go there", shows it for programs. The second is what you implement.

Step 6 — Let each side write the tests for the other. The person who *uses* a contract knows best what it must do, so the client owner writes tests for the server and the server owner writes tests for the client, each against a *stub*: a tiny fake of the other side.

Step 7 — Build a walking skeleton. Make the thinnest possible version of the whole system work end to end, one message from the window through the client and the network to the server and back, before making any part rich. Integration problems then appear on the third day instead of on the last one.

Step 8 — Add concurrency and failure handling on purpose. Decide which objects are shared between threads and how each is protected; decide what the client does when the server is down and what the server does when a client vanishes. Test both.

Step 9 — Measure, then document. Measure the numbers you will promise (how many clients, how fast), then write the user guide, the installation guide, the performance criteria and a troubleshooting table.

Step 10 — Write the report as you go. Keep a short log of decisions and of who did what. The report at the end is then a matter of editing, not of remembering.

Table 16.2 condenses the method; every problem in Sections 16.4 to 16.6 is one or two of its steps carried out.

Table 16.2 The ten-step method for a team project, and the problem of this chapter in which each step is carried out.

Step	What you produce	Problem of this chapter
1–3	one-sentence goal, user stories, parts with owners	16.1
4	contracts: interfaces and protocol	16.1, 16.2
5	flow chart, state machine, state table	16.2
6	cross tests with stubs	16.4
7	a walking skeleton, end to end	16.3
8	thread-safety and failure handling	16.5, 16.6, 16.7
9	measurements and product documentation	16.8
10	the project report	16.8

WHY CONTRACTS FIRST. Three people who agree on an interface on Monday can work in parallel all week, each with a stub for the parts they do not own. Three people who agree only "in words" discover on Friday that one of them numbered the jokes from 1 and another from 0. Problem 16.1 shows that exact bug and the test that catches it.

16.3 The Project Brief, Restated

The eight problems of the next three sections come from one project brief. Here it is in full, as a team would receive it.

THE BRIEF. Based on everything in this book, build a client–server application with a graphical user interface, in which the client and the server communicate through sockets. Examples: a game with a chat, a small "social network", a collaboration tool, an interactive portal, a start-up idea, or anything else the team finds exciting. Work in a team of at most three people. (a) Decide the topic and title of the project immediately. (b) Divide the work: each member writes the specification and the job description of their own part; the natural parts are the server, the client, the database and its connectivity, the graphical interface, and testing

> and documentation. (c) Describe the specification clearly and document every interface, where “interface” is a general term and need not be a Java `interface`. (d) Describe every protocol and flow chart clearly, for example a knock-knock protocol between server and client. (e) Think about test cases: if member A uses the functions of member B through an interface, A designs test cases for B, and B designs test cases for A. (f) Describe the design, from specification to design to implementation, and the product documentation: a user-level guide, performance criteria, installation, and troubleshooting. (g) Develop and use test plans, for example with stubs, multithreaded tests and distributed tests. (h) Take care that the client knows when the server is down, that the server knows when a client is down, and that connection errors on both sides are handled as exceptions. (i) Write a report on what was done, the output, the answers to these points, pseudocode and anything else that matters, including each member’s job description and share of the work.

The brief does not name a topic, so we choose one that is small enough to solve completely on paper and still has every part the brief asks for: the **Knock-Knock Club**, a server that tells knock-knock jokes to any number of clients at once, a console client, a window client, and a file of jokes that plays the role of the database. The knock-knock conversation is the example protocol the brief itself suggests, and Chapter 15 introduced it; here we build it properly. A real team would choose a richer topic, a chat, a game or the laundry system of Problem 16.14, and every solution below carries over unchanged.

Table 16.3 maps the demands of the brief to the problems that satisfy them.

Table 16.3 How the demands of the project brief are divided among Problems 16.1 to 16.8.

Demand of the brief	Problem	What we deliver
(a), (b) topic, division of work	16.1	topic, user stories, a table of parts, owners and contracts
(c) specification and interfaces	16.1	`JokeStore`, two implementations, one contract test
(d) protocol and flow chart	16.2	flow chart, state machine, `KnockKnockProtocol`
the server and client themselves	16.3	a multi-client server and a console client
(e) A tests B, B tests A	16.4	cross tests with stubs, no network needed
(g) multithreaded and distributed tests	16.5	twenty concurrent clients against a real server
(h) server down, client down	16.6	four failure scenarios, each detected and reported
the graphical interface	16.7	a Swing window and a Tkinter window, MVC, headless tests
(f), (i) documentation, performance, report	16.8	measured criteria, user guide, troubleshooting, report outline

16.4 Problems on Planning and Protocol Design

16.4.1 Problem 16.1 — A topic, a division of work, and the first contract

> **PROBLEM.** Choose the topic and title of the project, divide the work among the members, and write the specification and job description of each part. Document the interface between the storage part and the rest of the system so precisely that another member can implement it, and show with a test that two different implementations keep the same promise.

In our own words. Before anyone writes a server, the team must agree on three things: what the program is for, who builds which part, and what exactly each part promises the others. The last one is the one teams skip, and it is the one that matters most, because it is the only thing that lets people work at the same time. We will write the first promise, between the “database” and

everyone else, as an interface, and we will write a test that any implementation must pass. Then we will watch that test catch a realistic misunderstanding.

Design. Steps 1 to 3 of the method give the following. *Goal:* "A user connects to the Knock-Knock Club, hears a joke, answers in the right places, and may ask for another." *Not a goal:* accounts, scores or jokes written by users. *User stories:* connect; hear "Knock! Knock!"; answer "Who's there?"; hear the clue; answer "*clue* who?"; hear the punch line and "Want another? (y/n)"; answer `y` for another joke or `n` to leave; see a clear message if the server is not there. **Figure 16.1** shows the parts, their owners and the contracts between them, and **Table 16.4** is the specification and job description of each part, one row per member role.

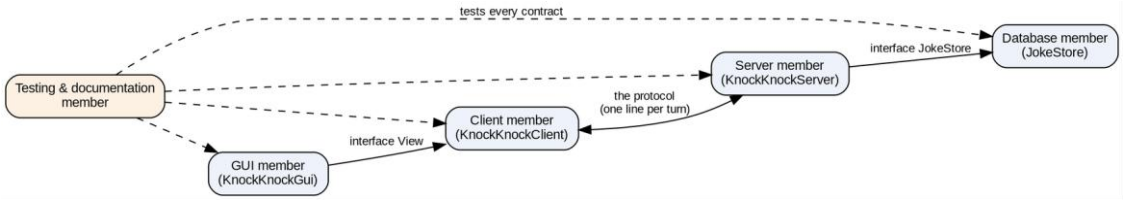


Figure 16.1 The five parts of the Knock-Knock Club and the contracts between them. Solid arrows are "uses"; dashed arrows show that the testing member tests every contract.

Table 16.4 The specification and job description of each part of the Knock-Knock Club. With three members, members 2 and 3 own two parts each.

Part (owner)	Specification	Contract it offers	Contract it uses
Database (member 1)	stores jokes; loads them from `jokes.txt`	`JokeStore`: `size()`, `get(i)`	none
Server (member 2)	accepts many clients, one conversation each	the knock-knock protocol	`JokeStore`
Client (member 2)	runs one conversation for one user	`converse(reader, writer, user)`	the protocol
GUI (member 3)	a window showing the conversation, never frozen	`View`: `showLine`, `setStatus`	the client
Testing and documentation (member 3)	contract tests, failure tests, user guide	none	all of them

The storage contract is the smallest one, so we write it first. A store holds at least one joke, the jokes are numbered from 0 to `size() - 1`, and asking for any other number throws `IndexOutOfBoundsException` (Python: `IndexError`). Writing the numbering rule down looks pedantic. It is the whole point.

The Java solution. **Listing 16.1** is the contract itself. It contains the `Joke` record (a small immutable value, Chapter 13), the two abstract methods with their documentation, and a static factory `of` that builds an in-memory store: the *stub* that every other member can use from the first hour, before the real database exists.

Listing 16.1 `JokeStore.java`: the storage contract, with the `Joke` record and an in-memory stub built by `JokeStore.of`.

```
// JokeStore.java - the contract between the database member and everybody else.
import java.util.List;

public interface JokeStore {

    /** One knock-knock joke: the clue ("Lettuce") and the answer ("Lettuce in, ..."). */
    record Joke(String clue, String answer) {
    }

    /** @return how many jokes the store holds; always at least 1. */
    int size();

    /**
     * Returns joke number index, counting from 0.
     *
     * @throws IndexOutOfBoundsException if index is not in 0 .. size()-1
     */
    Joke get(int index);

    /** An in-memory store: the stub every other member can use from day one. */
    static JokeStore of(Joke... jokes) {
        if (jokes.length == 0) {
            throw new IllegalArgumentException("a store needs at least one joke");
        }
        List<Joke> copy = List.of(jokes);               // an unmodifiable copy
        return new JokeStore() {
            @Override
            public int size() {
                return copy.size();
            }

            @Override
            public Joke get(int index) {
                return copy.get(index);                 // List.get checks the index
            }
        };
    }

    /** The three jokes used by the examples of this chapter. */
    static JokeStore classics() {
        return of(new Joke("Lettuce", "Lettuce in, it's cold out here!"),
                  new Joke("Harry", "Harry up, the laundry is ready!"),
                  new Joke("Olive", "Olive the clean shirts, thank you!"));
    }
}
```

Listing 16.2 is the database member's real work: a store that reads the file `jokes.txt`, a second store with a realistic mistake in it, and a *contract test*, one method that checks the promise and can be run against any implementation. The sample file has a comment, a blank line and four jokes:

```
# clue;answer
Lettuce;Lettuce in, it's cold out here!
Harry;Harry up, the laundry is ready!

Olive;Olive the clean shirts, thank you!
Iva;Iva sore hand from knocking!
```

Listing 16.2 `JokeStores.java`: a file-backed `JokeStore`, a one-based store that breaks the contract, and one contract test run against all three stores.

```
// JokeStores.java - Problem 16.1: two implementations of JokeStore and one contract test.
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

public class JokeStores {

    /** The database member's real store: reads "clue;answer" lines from a text file. */
    static final class FileJokeStore implements JokeStore {
        private final List<Joke> jokes;

        FileJokeStore(Path file) throws IOException {
            List<Joke> loaded = new ArrayList<>();
            int lineNo = 0;
            for (String line : Files.readAllLines(file)) {
                lineNo++;
                if (line.isBlank() || line.startsWith("#")) {
                    continue;                                   // comment or empty line
                }
                String[] parts = line.split(";", 2);            // at most two pieces
                if (parts.length != 2 || parts[0].isBlank()) {
                    throw new IOException(file + " line " + lineNo + ": expected
clue;answer");
                }
                loaded.add(new Joke(parts[0].trim(), parts[1].trim()));
            }
            if (loaded.isEmpty()) {
                throw new IOException(file + " contains no jokes");
            }
            this.jokes = List.copyOf(loaded);                   // freeze the list
        }

        @Override
        public int size() {
            return jokes.size();
        }

        @Override
        public Joke get(int index) {
            return jokes.get(index);
        }
    }
```

Listing 16.2 (Continued) *JokeStores.java: a file-backed JokeStore, a one-based store that breaks the contract, and one contract test run against all three stores.*

```
    /** Tempting but wrong: the member thought "joke number 1" is the first joke. */
    static final class OneBasedFileJokeStore implements JokeStore {
        private final FileJokeStore inner;

        OneBasedFileJokeStore(Path file) throws IOException {
            inner = new FileJokeStore(file);
        }

        @Override
        public int size() {
            return inner.size();
        }

        @Override
        public Joke get(int index) {
            return inner.get(index - 1);                        // off by one
        }
    }

    /** The contract, written ONCE, run against every implementation. Returns the failures.
*/
    static List<String> contract(JokeStore s) {
        List<String> failures = new ArrayList<>();
        if (s.size() < 1) {
            failures.add("size() must be at least 1");
        }
        try {
            if (s.get(0) == null || s.get(0).clue().isBlank()) {
                failures.add("get(0) must be a real joke");
            }
        } catch (RuntimeException e) {
            failures.add("get(0) threw " + e.getClass().getSimpleName());
        }
        for (int bad : new int[] {-1, s.size()}) {
            try {
                s.get(bad);
                failures.add("get(" + bad + ") must throw IndexOutOfBoundsException");
            } catch (IndexOutOfBoundsException expected) {
                // exactly what the contract promises
            }
        }
        return failures;
    }

    private static int passed = 0;
```

Listing 16.2 (Continued) *JokeStores.java: a file-backed JokeStore, a one-based store that breaks the contract, and one contract test run against all three stores.*

```
    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) throws IOException {
        Path file = Path.of("jokes.txt");
        JokeStore stub = JokeStore.classics();
        JokeStore real = new FileJokeStore(file);
        JokeStore wrong = new OneBasedFileJokeStore(file);

        System.out.println("stub : " + stub.size() + " jokes, first = " +
stub.get(0).clue());
        System.out.println("file : " + real.size() + " jokes, first = " +
real.get(0).clue());
        System.out.println("wrong: " + contract(wrong));

        check("stub keeps the contract", List.of(), contract(stub));
        check("file store keeps the contract", List.of(), contract(real));
        check("one-based store is caught", 2, contract(wrong).size());
        try {
            new FileJokeStore(Path.of("no_such_file.txt"));
            throw new AssertionError("no exception");
        } catch (IOException e) {
            check("missing file", "NoSuchFileException", e.getClass().getSimpleName());
        }
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
stub : 3 jokes, first = Lettuce
file : 4 jokes, first = Lettuce
wrong: [get(0) threw ArrayIndexOutOfBoundsException, get(4) must throw
IndexOutOfBoundsException]
PASS stub keeps the contract
PASS file store keeps the contract
PASS one-based store is caught
PASS missing file
All 4 checks passed.
```

The Python solution. Python has no `interface` keyword, so the contract is an abstract base class (Chapter 6). **Listing 16.3** is `joke_store.py`, the module every other Python file imports, and **Listing 16.4** holds the file store, the broken store and the same contract.

Listing 16.3 `joke_store.py`: the storage contract as an abstract base class, the frozen `Joke` dataclass and the in-memory stub `ListJokeStore`.

```
# joke_store.py - the contract between the database member and everybody else.
from abc import ABC, abstractmethod
from dataclasses import dataclass


@dataclass(frozen=True)
class Joke:
    """One knock-knock joke: the clue ("Lettuce") and the answer."""
    clue: str
    answer: str


class JokeStore(ABC):
    """A source of jokes, numbered from 0."""

    @abstractmethod
    def size(self):
        """How many jokes the store holds; always at least 1."""

    @abstractmethod
    def get(self, index):
        """Joke number index; raises IndexError unless 0 <= index < size()."""


class ListJokeStore(JokeStore):
    """An in-memory store: the stub every other member can use from day one."""

    def __init__(self, *jokes):
        if not jokes:
            raise ValueError("a store needs at least one joke")
        self._jokes = tuple(jokes)                  # an unchangeable copy

    def size(self):
        return len(self._jokes)

    def get(self, index):
        if not 0 <= index < len(self._jokes):      # a tuple would accept -1!
            raise IndexError(f"joke {index} does not exist")
        return self._jokes[index]


def classics():
    """The three jokes used by the examples of this chapter."""
    return ListJokeStore(Joke("Lettuce", "Lettuce in, it's cold out here!"),
                         Joke("Harry", "Harry up, the laundry is ready!"),
                         Joke("Olive", "Olive the clean shirts, thank you!"))
```

Listing 16.4 `joke_stores.py`: the file store, the one-based store and the contract, with `unittest` tests.

```
# joke_stores.py - Problem 16.1: two implementations of JokeStore and one contract test.
import sys
import unittest
from joke_store import Joke, JokeStore, ListJokeStore, classics


class FileJokeStore(JokeStore):
    """The database member's real store: reads "clue;answer" lines from a text file."""

    def __init__(self, file_name):
        loaded = []
        with open(file_name, encoding="utf-8") as f:
            for line_no, line in enumerate(f, start=1):
                line = line.strip()
                if not line or line.startswith("#"):
                    continue                            # comment or empty line
                parts = line.split(";", 1)              # at most two pieces
                if len(parts) != 2 or not parts[0].strip():
                    raise OSError(f"{file_name} line {line_no}: expected clue;answer")
                loaded.append(Joke(parts[0].strip(), parts[1].strip()))
        if not loaded:
            raise OSError(f"{file_name} contains no jokes")
        self._inner = ListJokeStore(*loaded)

    def size(self):
        return self._inner.size()

    def get(self, index):
        return self._inner.get(index)


class OneBasedFileJokeStore(FileJokeStore):
    """Tempting but wrong: the member thought "joke number 1" is the first joke."""

    def get(self, index):
        return super().get(index - 1)                   # off by one
```

Listing 16.4 (Continued) *joke_stores.py: the file store, the one-based store and the contract, with unittest tests.*

```
def contract(store):
    """The contract, written ONCE, run against every implementation."""
    failures = []
    if store.size() < 1:
        failures.append("size() must be at least 1")
    try:
        if not store.get(0).clue.strip():
            failures.append("get(0) must be a real joke")
    except Exception as e:
        failures.append(f"get(0) raised {type(e).__name__}")
    for bad in (-1, store.size()):
        try:
            store.get(bad)
            failures.append(f"get({bad}) must raise IndexError")
        except IndexError:
            pass                                        # exactly what the contract
promises
    return failures


class TestStores(unittest.TestCase):
    def test_stub_keeps_the_contract(self):
        self.assertEqual([], contract(classics()))

    def test_file_store_keeps_the_contract(self):
        self.assertEqual([], contract(FileJokeStore("jokes.txt")))

    def test_one_based_store_is_caught(self):
        self.assertEqual(2, len(contract(OneBasedFileJokeStore("jokes.txt"))))

    def test_missing_file(self):
        with self.assertRaises(FileNotFoundError):
            FileJokeStore("no_such_file.txt")


if __name__ == "__main__":
    stub, real = classics(), FileJokeStore("jokes.txt")
    print("stub :", stub.size(), "jokes, first =", stub.get(0).clue)
    print("file :", real.size(), "jokes, first =", real.get(0).clue)
    print("wrong:", contract(OneBasedFileJokeStore("jokes.txt")))
    sys.stdout.flush()                                  # print before the test report
    unittest.main(verbosity=2)
```

```
stub : 3 jokes, first = Lettuce
file : 4 jokes, first = Lettuce
wrong: ['get(0) raised IndexError', 'get(4) must raise IndexError']
test_file_store_keeps_the_contract (__main__.TestStores.test_file_store_keeps_the_contract)
... ok
test_missing_file (__main__.TestStores.test_missing_file) ... ok
test_one_based_store_is_caught (__main__.TestStores.test_one_based_store_is_caught) ... ok
test_stub_keeps_the_contract (__main__.TestStores.test_stub_keeps_the_contract) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Line by line. In **Listing 16.1**, `record Joke(String clue, String answer)` declares a class with two private final fields, a constructor, the two accessors `clue()` and `answer()`, and `equals`, `hashCode` and `toString`, all written by the compiler. A record nested in an interface is automatically `static` and `public`, so other files write `JokeStore.Joke`. The Javadoc of `get` states the numbering rule and the exception; that comment *is* the specification. `static JokeStore`

`of(Joke... jokes)` takes any number of jokes (`...` makes a *varargs* parameter), copies them with `List.of`, which gives an unmodifiable list, and returns an *anonymous class*: a class without a name, written and instantiated in one expression, that implements the two methods. `copy.get(index)` already throws `IndexOutOfBoundsException` for a bad index, so the stub keeps the contract for free.

In **Listing 16.2**, `FileJokeStore` reads every line with `Files.readAllLines`, skips comments and blank lines exactly as the parser of Problem 8.16 did, and splits each line with `split(";", 2)`. The limit 2 matters: an answer that itself contains a semicolon stays in one piece. A malformed line or an empty file is reported as an `IOException` with the line number, because a broken jokes file is a problem with the environment, not with the program. `List.copyOf(loaded)` freezes the list, so nobody can add jokes to a store after it is built. The `contract` method checks four promises: at least one joke, a real joke at index 0, and an exception for the indices `-1` and `size()`. It collects failures in a list instead of stopping at the first one, so a report tells the implementer everything that is wrong at once. The output shows both correct stores keeping the contract and the wrong store failing it twice.

In Python, `@dataclass(frozen=True)` writes `__init__`, `__eq__`, `__hash__` and `__repr__` for `Joke` and forbids assignment to its fields. `ListJokeStore.get` contains a check that Java did not need: a tuple *accepts* negative indices (`t[-1]` is the last element), so without the explicit test `0 <= index < len(...)` the stub itself would silently break the contract for `-1`. This is a small, real example of a contract being stricter than the language, and of the contract test being worth writing even for a stub. `OneBasedFileJokeStore` inherits from `FileJokeStore` and overrides only `get`, which keeps the broken version as short as the mistake.

A tempting wrong solution. `OneBasedFileJokeStore` is the mistake of a member who thought of "joke number 1" as the first joke. Nothing in the language stops it: it compiles, it returns real jokes for most indices, and a quick manual test ("get me joke 1") even looks right. The contract test catches it at once, and precisely: `get(0)` throws, because it asks the inner list for index −1, and `get(4)` *does not* throw, because it quietly returns the last joke. The server member, who wrote `get(current)` with `current` starting at 0, would have crashed on the very first connection. The deeper mistake is agreeing "in words". A written contract with a test turns a Friday-night integration disaster into a failing check on Monday.

> **CHECK YOURSELF.** A fourth member offers `RandomJokeStore`, whose `get(i)` ignores `i` and returns a random joke. Does it pass the contract test? Should it?
>
> *Answer.* It fails: `get(-1)` and `get(size())` return a joke instead of throwing, so the contract reports two failures. It should fail, because the server relies on `get(i)` meaning "joke number `i`" in order to tell the jokes one after the other. If random jokes are wanted, the right place is the server's choice of `i`, not the store.

16.4.2 Problem 16.2 — The protocol as a flow chart and a state machine

> **PROBLEM.** Describe the protocol between the server and the client clearly, with a flow chart. The knock-knock conversation goes: server "Knock, knock"; client "Who's there?"; server a clue; client "*clue* who?"; server the punch line; server "Want more?"; if the client answers yes, the server knocks again; if no, the server terminates. Implement the server's side of this protocol as a class that can be tested without any network.

In our own words. A protocol is a conversation with rules. People understand it best as a flow chart; programs need it as a *state machine*: at every moment the server is in one of a few states, and each line from the client, together with the current state, decides the reply and the next state.

We will draw both, turn the state machine into a table, and implement the table as a class with one method, `processInput`, that takes the client's line and returns the server's line. Because that class never touches a socket, it can be tested with plain strings.

Design. **Figure 16.2** is the flow chart of the brief. It is a good picture of the happy path, but it leaves out every question a program must answer: what if the client says "Who is it?" instead of "Who's there?", or gives the wrong clue back, or answers "maybe"?

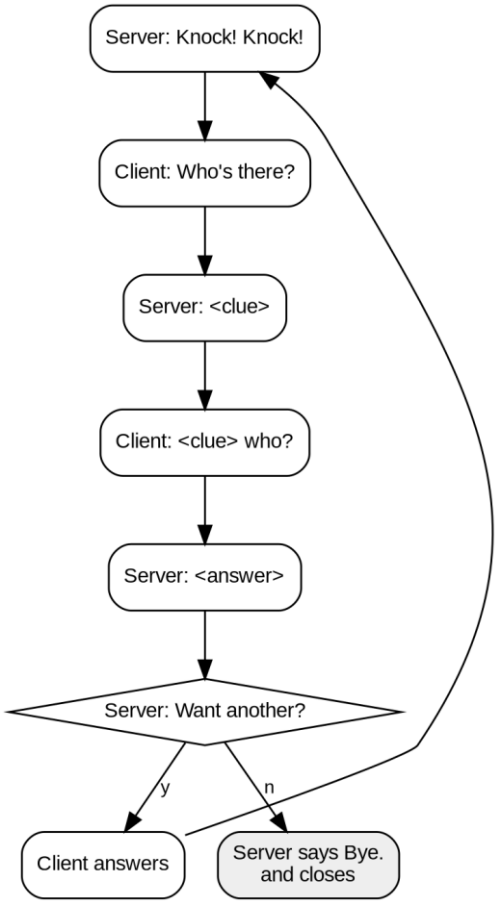


Figure 16.2 The knock-knock protocol as a flow chart: the conversation for people, with only the expected answers.

Figure 16.3 is the same protocol as a state machine, with every input covered. Each arrow is labeled "input / output": for example, in state `SENT_KNOCK`, the input "Who's there?" produces the output *clue* and leads to state `SENT_CLUE`. The rule for a wrong line is our design decision: the server politely restarts the current joke.

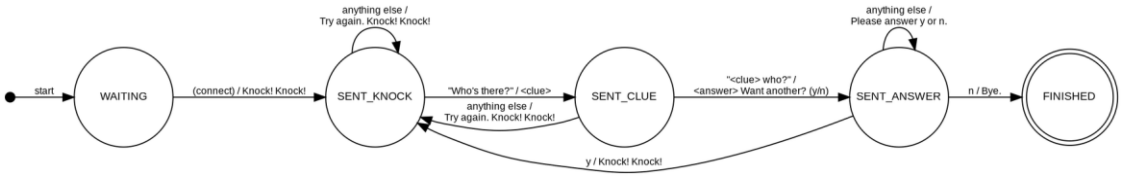


Figure 16.3 The same protocol as a state machine. Every state has an arrow for every possible input, including "anything else".

> **DEFINITION.** A *finite state machine* is a set of states, a start state, and a *transition function* that maps a pair (state, input) to a pair (output, next state). It is the standard way to specify a protocol, because it forces an answer for every input in every state.

Written as a function, the machine is

$$\delta(s, x) = (y, s') \tag{16.1}$$

where s is the current state, x the client's line, y the server's reply and s' the next state. Equation (16.1) is exactly the signature of `processInput`: its argument is x, its return value is y, and it updates the field `state` from s to s'. **Table 16.5** lists every transition; it is the specification the implementation must match line by line.

Table 16.5 The transition table of the knock-knock protocol. Every state has a row for "anything else", which the flow chart does not show.

State s	Client line x	Server reply y	Next state s'
`WAITING`	(none: the client has just connected)	`Knock! Knock!`	`SENT_KNOCK`
`SENT_KNOCK`	`Who's there?` (any case)	the clue	`SENT_CLUE`
`SENT_KNOCK`	anything else	`You're supposed to say "Who's there?"! Try again. Knock! Knock!`	`SENT_KNOCK`
`SENT_CLUE`	*clue* `who?` (any case)	the answer and `Want another? (y/n)`	`SENT_ANSWER`
`SENT_CLUE`	anything else	`You're supposed to say "`*clue*`who?"! Try again. Knock! Knock!`	`SENT_KNOCK`
`SENT_ANSWER`	`y`	`Knock! Knock!` (next joke)	`SENT_KNOCK`
`SENT_ANSWER`	`n`	`Bye.`	`FINISHED`
`SENT_ANSWER`	anything else	`Please answer y or n.`	`SENT_ANSWER`

One more decision belongs in the protocol: *framing*, the rule that tells the receiver where one message ends. Ours is the simplest possible: every message is exactly one line of text, ended by a newline, and the server always answers one line for each line it receives. That is why the punch line and "Want another? (y/n)" travel together in one line, although the flow chart draws them as two boxes. With one line per turn, a client can never be confused about whose turn it is.

The Java solution. **Listing 16.5** implements **Table 16.5** with an `enum` for the states and a `switch` on the current state.

Listing 16.5 `KnockKnockProtocol.java`: the transition table as a class, a stateless wrong version, and a `play` helper that runs a conversation without a network.

```java
// KnockKnockProtocol.java - Problem 16.2: the knock-knock flow chart as a state machine.
import java.util.ArrayList;
import java.util.List;

public class KnockKnockProtocol {

    /** The four boxes of the flow chart where the server waits, plus the end. */
    enum State { WAITING, SENT_KNOCK, SENT_CLUE, SENT_ANSWER, FINISHED }

    static final String KNOCK = "Knock! Knock!";
    static final String ANOTHER = " Want another? (y/n)";
    static final String BYE = "Bye.";

    private final JokeStore jokes;
    private State state = State.WAITING;
    private int current = 0;                        // index of the joke being told

    public KnockKnockProtocol(JokeStore jokes) {
        this.jokes = jokes;
    }

    public boolean finished() {
        return state == State.FINISHED;
    }

    /**
     * Given what the client just said (null before the first line), returns
     * what the server says next. One call is one arrow of the flow chart.
     */
    public String processInput(String input) {
        JokeStore.Joke joke = jokes.get(current);
        switch (state) {
            case WAITING:
                state = State.SENT_KNOCK;
                return KNOCK;
            case SENT_KNOCK:
                if ("Who's there?".equalsIgnoreCase(input)) {
                    state = State.SENT_CLUE;
                    return joke.clue();
                }
                return "You're supposed to say \"Who's there?\"! Try again. " + KNOCK;
            case SENT_CLUE:
                if ((joke.clue() + " who?").equalsIgnoreCase(input)) {
                    state = State.SENT_ANSWER;
                    return joke.answer() + ANOTHER;
                }
```

Listing 16.5 (Continued) *KnockKnockProtocol.java: the transition table as a class, a stateless wrong version, and a play helper that runs a conversation without a network.*

```
                state = State.SENT_KNOCK;
                return "You're supposed to say \"" + joke.clue() + " who?\"! Try again. " +
KNOCK;
            case SENT_ANSWER:
                if ("y".equalsIgnoreCase(input)) {
                    current = (current + 1) % jokes.size();     // next joke, wrap around
                    state = State.SENT_KNOCK;
                    return KNOCK;
                }
                if ("n".equalsIgnoreCase(input)) {
                    state = State.FINISHED;
                    return BYE;
                }
                return "Please answer y or n.";
            default:
                throw new IllegalStateException("the conversation is over");
        }
    }

    /** Tempting but wrong: no state at all, the server only looks at the last line. */
    static String statelessReply(String input) {
        if (input == null || input.equalsIgnoreCase("y")) {
            return KNOCK;
        }
        if (input.equalsIgnoreCase("Who's there?")) {
            return "Lettuce";
        }
        if (input.endsWith(" who?")) {
            return "Lettuce in, it's cold out here!" + ANOTHER;
        }
        return BYE;
    }

    /** Plays a whole conversation: the client's lines in, the transcript out. */
    static List<String> play(KnockKnockProtocol p, String... clientLines) {
        List<String> transcript = new ArrayList<>();
        transcript.add("S: " + p.processInput(null));
        for (String line : clientLines) {
            transcript.add("C: " + line);
            transcript.add("S: " + p.processInput(line));
        }
        return transcript;
    }

    private static int passed = 0;
```

Listing 16.5 (Continued) *KnockKnockProtocol.java: the transition table as a class, a stateless wrong version, and a play helper that runs a conversation without a network.*

```
    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        KnockKnockProtocol p = new KnockKnockProtocol(JokeStore.classics());
        List<String> t = play(p, "Who's there?", "Lettuce who?", "y",
                              "who's there?", "Harry who?", "n");
        t.forEach(System.out::println);
        check("finished after n", true, p.finished());
        check("second joke told", "S: Harry", t.get(8));

        KnockKnockProtocol q = new KnockKnockProtocol(JokeStore.classics());
        List<String> u = play(q, "Who is it?", "Who's there?", "Cabbage who?");
        check("wrong greeting", "S: You're supposed to say \"Who's there?\"! Try again.
Knock! Knock!", u.get(2));
        check("wrong clue", "S: You're supposed to say \"Lettuce who?\"! Try again. Knock!
Knock!", u.get(6));
        check("not finished", false, q.finished());
        check("stateless reply is fooled", "Lettuce in, it's cold out here! Want another?
(y/n)",
              statelessReply("Cabbage who?"));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
S: Knock! Knock!
C: Who's there?
S: Lettuce
C: Lettuce who?
S: Lettuce in, it's cold out here! Want another? (y/n)
C: y
S: Knock! Knock!
C: who's there?
S: Harry
C: Harry who?
S: Harry up, the laundry is ready! Want another? (y/n)
C: n
S: Bye.
PASS finished after n
PASS second joke told
PASS wrong greeting
PASS wrong clue
PASS not finished
PASS stateless reply is fooled
All 6 checks passed.
```

The Python solution. **Listing 16.6** is the same machine. Python's `match` statement would also work on Python 3.10 and later, but a chain of `if` statements on an `Enum` reads just as clearly and runs everywhere.

Listing 16.6 `knock_knock_protocol.py`: the same state machine in Python, with the same tests.

```
# knock_knock_protocol.py - Problem 16.2: the knock-knock flow chart as a state machine.
import sys
import unittest
from enum import Enum, auto
from joke_store import classics

KNOCK = "Knock! Knock!"
ANOTHER = " Want another? (y/n)"
BYE = "Bye."


class State(Enum):
    """The four boxes of the flow chart where the server waits, plus the end."""
    WAITING = auto()
    SENT_KNOCK = auto()
    SENT_CLUE = auto()
    SENT_ANSWER = auto()
    FINISHED = auto()


class KnockKnockProtocol:
    def __init__(self, jokes):
        self._jokes = jokes
        self._state = State.WAITING
        self._current = 0                           # index of the joke being told

    def finished(self):
        return self._state is State.FINISHED

    def process_input(self, text):
        """Given what the client just said (None before the first line), returns
        what the server says next. One call is one arrow of the flow chart."""
        joke = self._jokes.get(self._current)
        said = (text or "").lower()
        if self._state is State.WAITING:
            self._state = State.SENT_KNOCK
            return KNOCK
        if self._state is State.SENT_KNOCK:
            if said == "who's there?":
                self._state = State.SENT_CLUE
                return joke.clue
            return f"You're supposed to say \"Who's there?\"! Try again. {KNOCK}"
        if self._state is State.SENT_CLUE:
            if said == f"{joke.clue} who?".lower():
                self._state = State.SENT_ANSWER
                return joke.answer + ANOTHER
```

Listing 16.6 (Continued) *knock_knock_protocol.py: the same state machine in Python, with the same tests.*

```
            self._state = State.SENT_KNOCK
            return f"You're supposed to say \"{joke.clue} who?\"! Try again. {KNOCK}"
        if self._state is State.SENT_ANSWER:
            if said == "y":
                self._current = (self._current + 1) % self._jokes.size()   # wrap around
                self._state = State.SENT_KNOCK
                return KNOCK
            if said == "n":
                self._state = State.FINISHED
                return BYE
            return "Please answer y or n."
        raise RuntimeError("the conversation is over")


def stateless_reply(text):
    """Tempting but wrong: no state at all, the server only looks at the last line."""
    if text is None or text.lower() == "y":
        return KNOCK
    if text.lower() == "who's there?":
        return "Lettuce"
    if text.endswith(" who?"):
        return "Lettuce in, it's cold out here!" + ANOTHER
    return BYE


def play(protocol, *client_lines):
    """Plays a whole conversation: the client's lines in, the transcript out."""
    transcript = ["S: " + protocol.process_input(None)]
    for line in client_lines:
        transcript.append("C: " + line)
        transcript.append("S: " + protocol.process_input(line))
    return transcript


class TestProtocol(unittest.TestCase):
    def test_two_jokes_then_bye(self):
        p = KnockKnockProtocol(classics())
        t = play(p, "Who's there?", "Lettuce who?", "y", "who's there?", "Harry who?", "n")
        self.assertTrue(p.finished())
        self.assertEqual("S: Harry", t[8])
```

Listing 16.6 (Continued) *knock_knock_protocol.py: the same state machine in Python, with the same tests.*

```
    def test_wrong_lines_restart_the_joke(self):
        q = KnockKnockProtocol(classics())
        u = play(q, "Who is it?", "Who's there?", "Cabbage who?")
        self.assertEqual("S: You're supposed to say \"Who's there?\"! Try again. Knock!
Knock!", u[2])
        self.assertEqual("S: You're supposed to say \"Lettuce who?\"! Try again. Knock!
Knock!", u[6])
        self.assertFalse(q.finished())

    def test_stateless_reply_is_fooled(self):
        self.assertEqual("Lettuce in, it's cold out here! Want another? (y/n)",
                         stateless_reply("Cabbage who?"))


if __name__ == "__main__":
    for line in play(KnockKnockProtocol(classics()), "Who's there?", "Lettuce who?", "y",
                     "who's there?", "Harry who?", "n"):
        print(line)
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
S: Knock! Knock!
C: Who's there?
S: Lettuce
C: Lettuce who?
S: Lettuce in, it's cold out here! Want another? (y/n)
C: y
S: Knock! Knock!
C: who's there?
S: Harry
C: Harry who?
S: Harry up, the laundry is ready! Want another? (y/n)
C: n
S: Bye.
test_stateless_reply_is_fooled (__main__.TestProtocol.test_stateless_reply_is_fooled) ...
ok
test_two_jokes_then_bye (__main__.TestProtocol.test_two_jokes_then_bye) ... ok
test_wrong_lines_restart_the_joke (__main__.TestProtocol.test_wrong_lines_restart_the_joke)
... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Line by line. The `enum State` names the five states of **Figure 16.3**; an enum is better than an `int` or a `String` because the compiler rejects any state that does not exist. The two fields `state` and `current` are the whole memory of the conversation: which box we are in and which joke we are telling. `processInput` first fetches the current joke, then switches on the state; each `case` is one group of rows of **Table 16.5**. `"Who's there?".equalsIgnoreCase(input)` puts the constant first, so a `null` input returns `false` instead of throwing a `NullPointerException`. The `y` branch computes `(current + 1) % jokes.size()`: the remainder operator makes the joke number wrap from the last joke back to 0, so a user can ask for more jokes than the store holds. The `default` branch is reached only in `FINISHED`; calling the protocol after “Bye.” is a programming error, so it throws `IllegalStateException`. The `play` helper feeds a list of client lines through the protocol and records the transcript, with `S:` for the server and `C:` for the client. The first test plays two jokes and checks that the ninth line is the second clue; the second test sends a wrong greeting and a wrong clue and checks both polite restarts. The Python version lowercases the input once (`said`) and uses `(text or "")` so that `None` becomes an empty string, the equivalent of Java’s constant-first comparison.

A tempting wrong solution. `statelessReply` has no fields at all; it looks only at the last line. It is shorter, and it even handles the happy path. But without state it cannot know *which* clue it told, so it answers the punch line to any line that ends in ” who?“: the last check in each program shows it replying”Lettuce in, it’s cold out here!” to “Cabbage who?”. It also cannot tell a second joke, because it does not remember which joke came first. Whenever the right reply depends on what was said before, the object needs state, and the state machine is the design that makes that state explicit.

> **CHECK YOURSELF.** Add a rule: in state `SENT_KNOCK`, the input `quit` ends the conversation at once with the reply `Bye.`. Which row do you add to **Table 16.5**, and which lines of code change?
>
> *Answer.* The row (`SENT_KNOCK`, `quit`, `Bye.`, `FINISHED`). In Java, inside `case SENT_KNOCK`, before the “Who’s there?” test: `if ("quit".equalsIgnoreCase(input)) { state = State.FINISHED; return BYE; }`. In Python, the same three lines inside the `SENT_KNOCK` branch. The client needs no change, because it already stops at `Bye.`.

16.5 Problems on Building and Testing the Client–Server System

16.5.1 Problem 16.3 — A server for many clients, and a console client

> **PROBLEM.** Build the server and the client of the application. The server listens on a port and must serve several clients at the same time, each in its own conversation. The client connects to the server, shows each line from the server to the user, and sends the user's replies. Show that a second client is served while a first client is still in the middle of its conversation.

In our own words. This is the walking skeleton of Step 7: the protocol of Problem 16.2 and the store of Problem 16.1, joined by real sockets. The server's main thread only waits for new connections; every accepted connection gets its own thread and its own `KnockKnockProtocol` object, so conversations cannot disturb each other. The client is a loop: read a line, show it, ask the user, send the answer, until `Bye.` or until the server hangs up.

Design. **Figure 16.4** shows the threads of the server. The *acceptor thread* runs `accept()` in a loop; `accept()` blocks until a client connects and then returns a new `Socket` for that client alone, as Chapter 15 described. The acceptor hands the socket to a *worker thread* and immediately goes back to `accept()`. Each worker owns a fresh protocol object. The only objects shared by all workers are the joke store, which is immutable and therefore safe to share without any lock (Chapter 13), and the counters of finished and interrupted conversations, which are `AtomicInteger`s and therefore safe to update from many threads (Chapter 14).

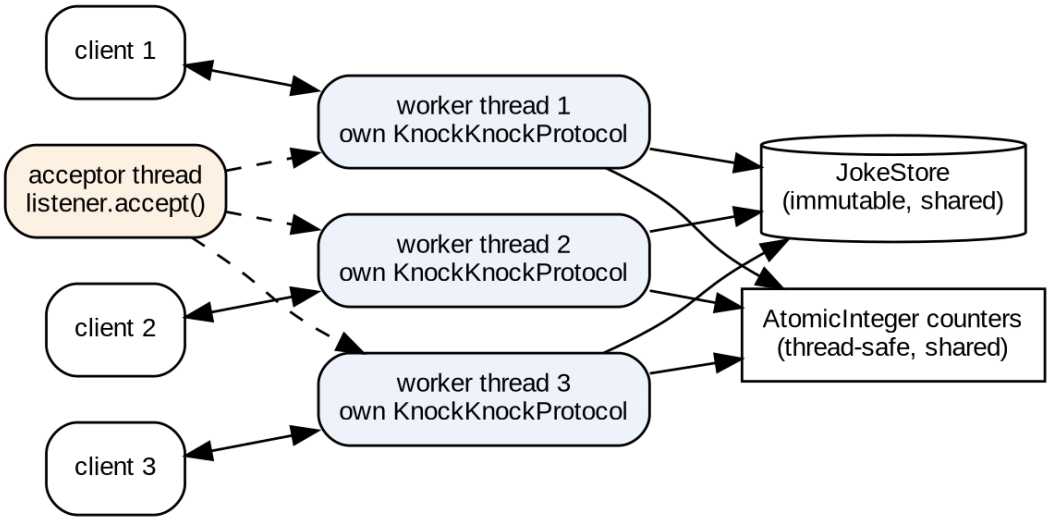


Figure 16.4 The threads of the knock-knock server. Each client has its own worker thread and its own protocol object; only immutable or thread-safe objects are shared.

The user of the client is modeled as a *function* from the server's line to the reply: `Function<String, String>` in Java, any callable in Python. A person at a keyboard is one such function; `PoliteUser`, which always answers correctly and wants a given number of jokes, is another. This small decision makes the whole client testable, because a test can replace the person with `PoliteUser`.

The Java solution. **Listing 16.7** is the server. Its `main` method runs it on port 4444, or on the port given on the command line.

Listing 16.7 `KnockKnockServer.java`: an acceptor thread, one worker per client, a static `converse` method that runs one conversation, and a deliberately wrong one-at-a-time mode.

```
// KnockKnockServer.java - Problem 16.3: a server that talks to many clients at once.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;

public class KnockKnockServer {

    private final JokeStore jokes;                  // immutable, so safe to share
    private final boolean oneAtATime;               // true only for the wrong variant
    private final ExecutorService workers = Executors.newCachedThreadPool();
    private final AtomicInteger finished = new AtomicInteger();
    private final AtomicInteger leftEarly = new AtomicInteger();
    private final List<String> log = new CopyOnWriteArrayList<>();
    private ServerSocket listener;

    public KnockKnockServer(JokeStore jokes) {
        this(jokes, false);
    }

    KnockKnockServer(JokeStore jokes, boolean oneAtATime) {
        this.jokes = jokes;
        this.oneAtATime = oneAtATime;
    }

    /** Opens the listening socket (port 0 means "any free port") and returns the port. */
    public int start(int port) throws IOException {
        listener = new ServerSocket(port);
        Thread acceptor = new Thread(this::acceptLoop, "acceptor");
        acceptor.setDaemon(true);                   // does not keep the program alive
        acceptor.start();
        return listener.getLocalPort();
    }
```

Listing 16.7 (Continued) *KnockKnockServer.java: an acceptor thread, one worker per client, a static converse method that runs one conversation, and a deliberately wrong one-at-a-time mode.*

```
    private void acceptLoop() {
        try {
            while (true) {
                Socket client = listener.accept();          // waits for the next client
                if (oneAtATime) {
                    serve(client);                          // wrong: everyone else waits
                } else {
                    workers.execute(() -> serve(client));   // one thread per conversation
                }
            }
        } catch (IOException e) {
            // stop() closed the listening socket, so accept() failed: the loop ends
        }
    }

    private void serve(Socket client) {
        try (client;
             BufferedReader in = new BufferedReader(new InputStreamReader(
                     client.getInputStream(), StandardCharsets.UTF_8));
             PrintWriter out = new PrintWriter(new OutputStreamWriter(
                     client.getOutputStream(), StandardCharsets.UTF_8), true)) {
            if (converse(in, out, new KnockKnockProtocol(jokes))) {
                finished.incrementAndGet();
            } else {
                leftEarly.incrementAndGet();
                log.add("client left before the end");
            }
        } catch (IOException e) {
            leftEarly.incrementAndGet();
            log.add("connection lost: " + e.getClass().getSimpleName());
        }
    }

    /** One conversation. Returns true if it reached "Bye.", false if the client vanished.
*/
    static boolean converse(BufferedReader in, PrintWriter out, KnockKnockProtocol
protocol)
            throws IOException {
        out.println(protocol.processInput(null));           // the server speaks first
        while (!protocol.finished()) {
            String line = in.readLine();
            if (line == null) {
                return false;                               // the client closed the
connection
            }
            out.println(protocol.processInput(line));
        }
        return true;
    }
```

Listing 16.7 (Continued) *KnockKnockServer.java: an acceptor thread, one worker per client, a static converse method that runs one conversation, and a deliberately wrong one-at-a-time mode.*

```
    public void stop() throws IOException {
        listener.close();
        workers.shutdown();
    }

    int finishedCount() {
        return finished.get();
    }

    int leftEarlyCount() {
        return leftEarly.get();
    }

    List<String> log() {
        return List.copyOf(log);
    }

    public static void main(String[] args) throws Exception {
        int port = (args.length > 0) ? Integer.parseInt(args[0]) : 4444;
        KnockKnockServer server = new KnockKnockServer(JokeStore.classics());
        System.out.println("Knock-knock server listening on port " + server.start(port));
        System.out.println("Press Ctrl+C to stop it.");
        Thread.currentThread().join();                      // wait forever
    }
}
```

Started from a terminal, the server prints two lines and then waits:

```
Knock-knock server listening on port 4444
Press Ctrl+C to stop it.
```

Listing 16.8 is the client. With two arguments it talks to a real server for a person at the keyboard; with no arguments it starts a server inside the same program on a free port and runs a self-test.

Listing 16.8 `KnockKnockClient.java`: the client loop, the scripted `PoliteUser`, an interactive mode and a self-test that also shows the one-at-a-time server starving a second client.

```
// KnockKnockClient.java - Problem 16.3: a client, a scripted user, and a self-test.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;

public class KnockKnockClient {

    /** A user who always answers correctly and wants a given number of jokes. */
    static final class PoliteUser implements Function<String, String> {
        private int jokesWanted;

        PoliteUser(int jokesWanted) {
            this.jokesWanted = jokesWanted;
        }

        @Override
        public String apply(String serverLine) {
            if (serverLine.endsWith("Knock! Knock!")) {
                return "Who's there?";
            }
            if (serverLine.endsWith("(y/n)")) {
                jokesWanted--;
                return (jokesWanted > 0) ? "y" : "n";
            }
            return serverLine + " who?";                    // the clue: "<clue> who?"
        }
    }
```

Listing 16.8 (Continued) *KnockKnockClient.java: the client loop, the scripted PoliteUser, an interactive mode and a self-test that also shows the one-at-a-time server starving a second client.*

```
    /** Talks to a server through any reader and writer; returns the transcript. */
    static List<String> converse(BufferedReader in, PrintWriter out,
                                 Function<String, String> user) throws IOException {
        List<String> transcript = new ArrayList<>();
        String fromServer;
        while ((fromServer = in.readLine()) != null) {      // null: the server hung up
            transcript.add("S: " + fromServer);
            if (fromServer.equals("Bye.")) {
                break;
            }
            String reply = user.apply(fromServer);
            if (reply == null) {
                break;                                      // the user gave up
            }
            transcript.add("C: " + reply);
            out.println(reply);
        }
        return transcript;
    }

    /** Opens a socket, converses, and closes everything again. */
    static List<String> talkTo(String host, int port, Function<String, String> user)
            throws IOException {
        try (Socket socket = new Socket(host, port);
             BufferedReader in = new BufferedReader(new InputStreamReader(
                     socket.getInputStream(), StandardCharsets.UTF_8));
             PrintWriter out = new PrintWriter(new OutputStreamWriter(
                     socket.getOutputStream(), StandardCharsets.UTF_8), true)) {
            return converse(in, out, user);
        }
    }

    /** Connects, reads the first line with a time limit, and keeps the socket open. */
    static String firstLineWithin(int port, int millis, List<Socket> keepOpen) throws
IOException {
        Socket s = new Socket("localhost", port);
        keepOpen.add(s);
        s.setSoTimeout(millis);                             // readLine gives up after
millis
        BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));
        try {
            return in.readLine();
        } catch (SocketTimeoutException e) {
            return "(nothing within " + millis + " ms)";
        }
    }
```

Listing 16.8 (Continued) *KnockKnockClient.java: the client loop, the scripted PoliteUser, an interactive mode and a self-test that also shows the one-at-a-time server starving a second client.*

```
    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) throws Exception {
        if (args.length == 2) {                          // java KnockKnockClient
localhost 4444
            BufferedReader keyboard = new BufferedReader(new InputStreamReader(System.in));
            Function<String, String> human = line -> {
                System.out.println("Server: " + line);
                System.out.print("You: ");
                try {
                    return keyboard.readLine();
                } catch (IOException e) {
                    return null;
                }
            };
            List<String> t = talkTo(args[0], Integer.parseInt(args[1]), human);
            System.out.println("Server: " + t.get(t.size() - 1).substring(3));
            return;
        }
        KnockKnockServer server = new KnockKnockServer(JokeStore.classics());
        int port = server.start(0);                      // any free port
        List<String> t = talkTo("localhost", port, new PoliteUser(2));
        t.forEach(System.out::println);
        check("ends with Bye.", "S: Bye.", t.get(t.size() - 1));
        check("13 lines", 13, t.size());

        List<Socket> open = new ArrayList<>();
        firstLineWithin(port, 500, open);                // client A connects and stays
silent
        check("B served while A talks", "Knock! Knock!", firstLineWithin(port, 500, open));
        server.stop();
```

Listing 16.8 (Continued) *KnockKnockClient.java: the client loop, the scripted PoliteUser, an interactive mode and a self-test that also shows the one-at-a-time server starving a second client.*

```
        KnockKnockServer wrong = new KnockKnockServer(JokeStore.classics(), true);
        int wrongPort = wrong.start(0);
        firstLineWithin(wrongPort, 500, open);           // A again
        String b = firstLineWithin(wrongPort, 500, open);   // B must wait for A to finish
        System.out.println("one-at-a-time server, client B got: " + b);
        check("B starved by the wrong server", "(nothing within 500 ms)", b);
        for (Socket s : open) {
            s.close();
        }
        wrong.stop();
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
S: Knock! Knock!
C: Who's there?
S: Lettuce
C: Lettuce who?
S: Lettuce in, it's cold out here! Want another? (y/n)
C: y
S: Knock! Knock!
C: Who's there?
S: Harry
C: Harry who?
S: Harry up, the laundry is ready! Want another? (y/n)
C: n
S: Bye.
PASS ends with Bye.
PASS 13 lines
PASS B served while A talks
one-at-a-time server, client B got: (nothing within 500 ms)
PASS B starved by the wrong server
All 4 checks passed.
```

To use the programs as two people would, open two terminals in the `java` folder. In the first, type `java KnockKnockServer`; in the second, `java KnockKnockClient localhost 4444`, and answer the jokes. A session looks like this (after each `You:` the user types an answer, which the terminal shows on the same line):

```
Server: Knock! Knock!
You: Who's there?
Server: Lettuce
You: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
You: n
Server: Bye.
```

The Python solution. **Listing 16.9** and **Listing 16.10** have the same structure. Python's `socket.makefile` wraps a socket in a text file object, so reading a line is `reader.readline()` and writing one is `writer.write(...)` followed by `writer.flush()`.

Listing 16.9 `knock_knock_server.py`: the same server with `threading.Thread` workers and a lock around the counters.

```
# knock_knock_server.py - Problem 16.3: a server that talks to many clients at once.
import socket
import sys
import threading
from joke_store import classics
from knock_knock_protocol import KnockKnockProtocol


def send(writer, line):
    """Writes one line and pushes it onto the network at once."""
    writer.write(line + "\n")
    writer.flush()


def converse(reader, writer, protocol):
    """One conversation. Returns True if it reached "Bye.", False if the client
vanished."""
    send(writer, protocol.process_input(None))          # the server speaks first
    while not protocol.finished():
        line = reader.readline()
        if line == "":
            return False                                # the client closed the connection
        send(writer, protocol.process_input(line.rstrip("\r\n")))
    return True


class KnockKnockServer:
    def __init__(self, jokes, one_at_a_time=False):
        self._jokes = jokes                             # immutable, so safe to share
        self._one_at_a_time = one_at_a_time             # True only for the wrong variant
        self._lock = threading.Lock()                   # guards the three fields below
        self._finished = 0
        self._left_early = 0
        self._log = []
        self._listener = None

    def start(self, port):
        """Opens the listening socket (port 0 means "any free port") and returns the
port."""
        self._listener = socket.create_server(("localhost", port))
        threading.Thread(target=self._accept_loop, daemon=True).start()
        return self._listener.getsockname()[1]
```

Listing 16.9 (Continued) *knock_knock_server.py: the same server with threading.Thread workers and a lock around the counters.*

```
    def _accept_loop(self):
        try:
            while True:
                client, _address = self._listener.accept()     # waits for the next client
                if self._one_at_a_time:
                    self._serve(client)                         # wrong: everyone else
waits
                else:
                    threading.Thread(target=self._serve, args=(client,),
daemon=True).start()
        except OSError:
            pass                    # stop() closed the listening socket: the loop ends

    def _serve(self, client):
        try:
            with client, client.makefile("r", encoding="utf-8") as reader, \
                    client.makefile("w", encoding="utf-8") as writer:
                done = converse(reader, writer, KnockKnockProtocol(self._jokes))
            self._record(done, "client left before the end")
        except OSError as e:
            self._record(False, f"connection lost: {type(e).__name__}")

    def _record(self, done, message):
        with self._lock:
            if done:
                self._finished += 1
            else:
                self._left_early += 1
                self._log.append(message)

    def counts(self):
        with self._lock:
            return self._finished, self._left_early, list(self._log)

    def stop(self):
        try:
            self._listener.shutdown(socket.SHUT_RDWR)   # wakes up accept() on Linux
        except OSError:
            pass
        self._listener.close()
```

Listing 16.9 (Continued) *knock_knock_server.py: the same server with threading.Thread workers and a lock around the counters.*

```
if __name__ == "__main__":
    server = KnockKnockServer(classics())
    print("Knock-knock server listening on port",
          server.start(int(sys.argv[1]) if len(sys.argv) > 1 else 4444))
    print("Press Ctrl+C to stop it.")
    try:
        threading.Event().wait()                        # wait forever
    except KeyboardInterrupt:
        server.stop()
```

Listing 16.10 `knock_knock_client.py`: the client, `PoliteUser` as a callable class, and three tests including the starved second client.

```
# knock_knock_client.py - Problem 16.3: a client, a scripted user, and a self-test.
import socket
import sys
import unittest
from joke_store import classics
from knock_knock_server import KnockKnockServer, send


class PoliteUser:
    """A user who always answers correctly and wants a given number of jokes."""

    def __init__(self, jokes_wanted):
        self._jokes_wanted = jokes_wanted

    def __call__(self, server_line):
        if server_line.endswith("Knock! Knock!"):
            return "Who's there?"
        if server_line.endswith("(y/n)"):
            self._jokes_wanted -= 1
            return "y" if self._jokes_wanted > 0 else "n"
        return server_line + " who?"                    # the clue: "<clue> who?"


def converse(reader, writer, user):
    """Talks to a server through any reader and writer; returns the transcript."""
    transcript = []
    while True:
        line = reader.readline()
        if line == "":
            break                                       # the server hung up
        from_server = line.rstrip("\r\n")
        transcript.append("S: " + from_server)
        if from_server == "Bye.":
            break
        reply = user(from_server)
        if reply is None:
            break                                       # the user gave up
        transcript.append("C: " + reply)
        send(writer, reply)
    return transcript
```

Listing 16.10 (Continued) *knock_knock_client.py: the client, PoliteUser as a callable class, and three tests including the starved second client.*

```
def talk_to(host, port, user):
    """Opens a socket, converses, and closes everything again."""
    with socket.create_connection((host, port)) as s, \
            s.makefile("r", encoding="utf-8") as reader, \
            s.makefile("w", encoding="utf-8") as writer:
        return converse(reader, writer, user)


def first_line_within(port, seconds, keep_open):
    """Connects, reads the first line with a time limit, and keeps the socket open."""
    s = socket.create_connection(("localhost", port))
    keep_open.append(s)
    s.settimeout(seconds)                               # readline gives up after this
    try:
        return s.makefile("r", encoding="utf-8").readline().rstrip("\r\n")
    except TimeoutError:
        return f"(nothing within {int(seconds * 1000)} ms)"


class TestClientAndServer(unittest.TestCase):
    def test_two_jokes(self):
        server = KnockKnockServer(classics())
        t = talk_to("localhost", server.start(0), PoliteUser(2))
        server.stop()
        self.assertEqual("S: Bye.", t[-1])
        self.assertEqual(13, len(t))

    def test_second_client_is_served_at_once(self):
        server, open_sockets = KnockKnockServer(classics()), []
        port = server.start(0)
        first_line_within(port, 0.5, open_sockets)      # client A connects and stays
silent
        self.assertEqual("Knock! Knock!", first_line_within(port, 0.5, open_sockets))
        for s in open_sockets:
            s.close()
        server.stop()
```

Listing 16.10 (Continued) *knock_knock_client.py: the client, PoliteUser as a callable class, and three tests including the starved second client.*

```
    def test_one_at_a_time_server_starves_the_second_client(self):
        wrong, open_sockets = KnockKnockServer(classics(), one_at_a_time=True), []
        port = wrong.start(0)
        first_line_within(port, 0.5, open_sockets)
        b = first_line_within(port, 0.5, open_sockets)
        print("\none-at-a-time server, client B got:", b, end=" ... ", file=sys.stderr)
        self.assertEqual("(nothing within 500 ms)", b)
        for s in open_sockets:
            s.close()
        wrong.stop()


def human(line):
    print("Server:", line)
    try:
        return input("You: ")
    except EOFError:
        return None


if __name__ == "__main__":
    if len(sys.argv) == 3:                              # python knock_knock_client.py
localhost 4444
        print("Server:", talk_to(sys.argv[1], int(sys.argv[2]), human)[-1][3:])
    else:
        demo = KnockKnockServer(classics())
        for line in talk_to("localhost", demo.start(0), PoliteUser(2)):
            print(line)
        demo.stop()
        sys.stdout.flush()
        unittest.main(verbosity=2)
```

```
S: Knock! Knock!
C: Who's there?
S: Lettuce
C: Lettuce who?
S: Lettuce in, it's cold out here! Want another? (y/n)
C: y
S: Knock! Knock!
C: Who's there?
S: Harry
C: Harry who?
S: Harry up, the laundry is ready! Want another? (y/n)
C: n
S: Bye.
test_one_at_a_time_server_starves_the_second_client
(__main__.TestClientAndServer.test_one_at_a_time_server_starves_the_second_client) ...
one-at-a-time server, client B got: (nothing within 500 ms) ... ok
test_second_client_is_served_at_once
(__main__.TestClientAndServer.test_second_client_is_served_at_once) ... ok
test_two_jokes (__main__.TestClientAndServer.test_two_jokes) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.511s

OK
```

Line by line. In **Listing 16.7**, `start(int port)` creates the `ServerSocket`. Port 0 asks the operating system for any free port, which is what tests want: two test runs never fight over the same port, and `getLocalPort()` reports which port was chosen. The acceptor is a *daemon* thread, one that does not keep the Java program alive by itself, so a test can end without stopping it by hand. `acceptLoop` calls `listener.accept()`, then `workers.execute(() -> serve(client))`: the

lambda is a `Runnable` that the thread pool of `Executors.newCachedThreadPool()` runs on a thread it creates or reuses. In `serve`, the `try (client; BufferedReader in = ...; PrintWriter out = ...)` header closes the socket and both streams whatever happens; `client` can be listed there because it is effectively final. The `true` in `new PrintWriter(..., true)` switches on *autoflush*: every `println` pushes its bytes onto the network immediately. Problem 16.13 shows what happens without it. `converse` is `static` and takes a reader and a writer instead of a socket, so Problem 16.4 can test it with strings. It speaks first (`processInput(null)`), then loops until the protocol is finished, and returns `false` if `readLine()` returns `null`, which is how Java reports that the other side closed the connection.

In **Listing 16.8**, `converse` is the mirror image: read, record, stop at `Bye.`, ask the user, send. It stops also when `readLine()` returns `null` (the server hung up) and when the user function returns `null` (the person pressed Ctrl+D or Ctrl+Z). `talkTo` wraps `converse` in a `try`-with-resources that opens and closes the socket. `firstLineWithin` sets `setSoTimeout(millis)`, after which a `readLine()` that waits longer than `millis` milliseconds throws `SocketTimeoutException`; a test must never be able to wait forever. The self-test runs a two-joke conversation, checks that it has 13 lines and ends with `Bye.`, then connects a silent client A and checks that client B still gets "Knock! Knock!" within 500 ms.

The Python server uses `socket.create_server(("localhost", port))`, which creates, binds and starts listening in one call, and `threading.Thread(target=..., args=(client,), daemon=True)` for each worker. Because Python has no `AtomicInteger`, the counters live behind one `threading.Lock`, and `counts()` returns a consistent snapshot of all three under that lock. `stop()` calls `shutdown` before `close`, because on Linux a thread blocked in `accept()` is woken reliably only by `shutdown`.

A tempting wrong solution. The shortest server calls `serve(client)` directly inside the accept loop, which is what `oneAtATime` does. It works perfectly with one client, which is how most people test it. With two, the second client connects (the operating system completes the connection and queues it), but nobody reads from it or writes to it until the first conversation ends. The self-test shows exactly this: against the wrong server, client B hears nothing within 500 ms. A knock-knock user who goes to make coffee in the middle of a joke would block the whole club.

> **CHECK YOURSELF.** The server of **Listing 16.7** creates a new thread for every client. What happens if ten thousand clients connect at once, and how would you limit it?
>
> *Answer.* `newCachedThreadPool` creates up to ten thousand threads, each with its own stack of typically hundreds of kilobytes or more, which can exhaust memory. Use `Executors.newFixedThreadPool(200)` instead: at most 200 conversations run at once and the others wait in the pool's queue. (Recent Java versions also offer *virtual threads*, `Executors.newVirtualThreadPerTaskExecutor()`, which are cheap enough for one per client even in large numbers.) In Python, use a `ThreadPoolExecutor(max_workers=200)` and submit `self._serve`.

16.5.2 Problem 16.4 — Each side tests the other, with stubs

> **PROBLEM.** Think about test cases across the interface between two members. Member A uses the functions of member B through an interface, so A designs test cases for B; B uses A's functions, so B designs test cases for A. Apply this to the client and the server of the Knock-Knock Club, using stubs so that the tests need no running network.

In our own words. The person who *uses* a contract is the best person to test it, because they know what they rely on. Here the contract between client and server is the protocol. The client

owner writes tests that feed fixed client lines to the server's `converse` method and check its replies; the server owner writes tests that feed fixed server lines to the client's `converse` and check what the client says back. In both cases the other side is replaced by a *stub*.

> **DEFINITION.** A *stub* is a minimal stand-in for a component that the code under test depends on. It gives fixed answers and has no logic of its own. A test with a stub checks one component alone, quickly and without the rest of the system.

Design. **Figure 16.5** shows the arrangement. The stub on each side is the same tiny helper, `replay(lines)`, which turns a list of lines into a reader, as if the other side had sent exactly those lines. The replies are captured with a `StringWriter` (Python: `io.StringIO`). Because `converse` in Problem 16.3 takes a reader and a writer rather than a socket, no network is involved at all, and each test takes microseconds.

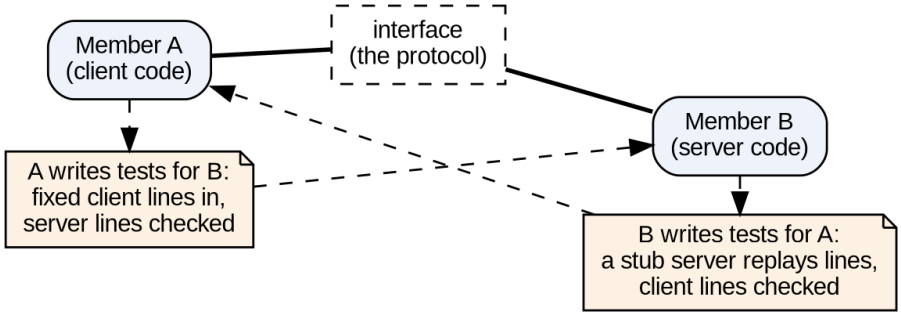


Figure 16.5 Cross testing through an interface. The client member writes tests for the server code; the server member writes tests for the client code. Each uses a stub of its own side.

The Java solution. **Listing 16.11** holds both sets of tests, and a *mutant*: a client with one deliberate mistake, used to check that the tests are strong enough.

Listing 16.11 `ProtocolContractTest.java`: tests written by each member for the other, with stubs that replay fixed lines, and a mutant client that a weak test lets through.

```
// ProtocolContractTest.java - Problem 16.4: each side tests the other through the
protocol, with stubs.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringReader;
import java.io.StringWriter;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;

public class ProtocolContractTest {

    /** A stub for either side: it just replays fixed lines, whatever the other side says.
*/
    static BufferedReader replay(String... lines) {
        return new BufferedReader(new StringReader(String.join("\n", lines) + "\n"));
    }

    /** The CLIENT member tests the SERVER: fixed client lines in, the server's lines out.
*/
    static List<String> serverSays(String... clientLines) throws IOException {
        StringWriter sent = new StringWriter();
        boolean done = KnockKnockServer.converse(replay(clientLines), new PrintWriter(sent,
true),
                                                 new
KnockKnockProtocol(JokeStore.classics()));
        List<String> lines = new ArrayList<>(sent.toString().lines().toList());
        lines.add(done ? "(finished)" : "(client left)");
        return lines;
    }

    /** The SERVER member tests the CLIENT: fixed server lines in, the client's lines out.
*/
    static List<String> clientSays(Function<String, String> user, String... serverLines)
            throws IOException {
        StringWriter sent = new StringWriter();
        KnockKnockClient.converse(replay(serverLines), new PrintWriter(sent, true), user);
        return sent.toString().lines().toList();
    }

    /** A mutant client: almost right, but it says "Who is there?". */
    static final class SloppyUser implements Function<String, String> {
        private final KnockKnockClient.PoliteUser polite = new
KnockKnockClient.PoliteUser(1);

        @Override
        public String apply(String line) {
            String reply = polite.apply(line);
            return reply.equals("Who's there?") ? "Who is there?" : reply;
        }
    }
```

Listing 16.11 (Continued) *ProtocolContractTest.java: tests written by each member for the other, with stubs that replay fixed lines, and a mutant client that a weak test lets through.*

```
    /** Tempting but wrong: a test that only checks that the client said something. */
    static boolean weakTest(Function<String, String> user) throws IOException {
        return !clientSays(user, "Knock! Knock!", "Bye.").isEmpty();
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) throws IOException {
        // Tests written by the client member for the server member.
        check("server speaks first", "Knock! Knock!", serverSays("Who's there?").get(0));
        check("one whole joke", List.of("Knock! Knock!", "Lettuce",
                "Lettuce in, it's cold out here! Want another? (y/n)", "Bye.",
"(finished)"),
                serverSays("Who's there?", "Lettuce who?", "n"));
        check("case does not matter", "Lettuce", serverSays("WHO'S THERE?").get(1));
        check("client vanishes", "(client left)", serverSays("Who's there?").get(2));

        // Tests written by the server member for the client member.
        String[] oneJoke = {"Knock! Knock!", "Lettuce",
                            "Lettuce in, it's cold out here! Want another? (y/n)", "Bye."};
        List<String> expected = List.of("Who's there?", "Lettuce who?", "n");
        check("client follows the protocol", expected,
              clientSays(new KnockKnockClient.PoliteUser(1), oneJoke));
        check("client stops at Bye.", List.of("Who's there?"),
              clientSays(new KnockKnockClient.PoliteUser(1), "Knock! Knock!", "Bye.",
"Knock! Knock!"));
        check("client survives a hang-up", List.of("Who's there?"),
              clientSays(new KnockKnockClient.PoliteUser(1), "Knock! Knock!"));

        // The mutant passes the weak test and fails the strong one.
        System.out.println("weak test passes the sloppy client: " + weakTest(new
SloppyUser()));
        check("strong test catches the sloppy client", false,
              expected.equals(clientSays(new SloppyUser(), oneJoke)));
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
PASS server speaks first
PASS one whole joke
PASS case does not matter
PASS client vanishes
PASS client follows the protocol
PASS client stops at Bye.
PASS client survives a hang-up
weak test passes the sloppy client: true
PASS strong test catches the sloppy client
All 8 checks passed.
```

The Python solution. **Listing 16.12** arranges the same tests in two `unittest` classes, one for each member who wrote them, so the report shows at a glance who promised what.

Listing 16.12 `protocol_contract_test.py`: the cross tests in Python, one test class per member.

```
# protocol_contract_test.py - Problem 16.4: each side tests the other through the protocol,
with stubs.
import io
import sys
import unittest
from joke_store import classics
from knock_knock_protocol import KnockKnockProtocol
import knock_knock_server
import knock_knock_client
from knock_knock_client import PoliteUser


def replay(*lines):
    """A stub for either side: it just replays fixed lines, whatever the other side
says."""
    return io.StringIO("\n".join(lines) + "\n")


def server_says(*client_lines):
    """The CLIENT member tests the SERVER: fixed client lines in, the server's lines
out."""
    sent = io.StringIO()
    done = knock_knock_server.converse(replay(*client_lines), sent,
KnockKnockProtocol(classics()))
    return sent.getvalue().splitlines() + ["(finished)" if done else "(client left)"]


def client_says(user, *server_lines):
    """The SERVER member tests the CLIENT: fixed server lines in, the client's lines
out."""
    sent = io.StringIO()
    knock_knock_client.converse(replay(*server_lines), sent, user)
    return sent.getvalue().splitlines()


class SloppyUser(PoliteUser):
    """A mutant client: almost right, but it says "Who is there?"."""

    def __call__(self, line):
        reply = super().__call__(line)
        return "Who is there?" if reply == "Who's there?" else reply


def weak_test(user):
    """Tempting but wrong: a test that only checks that the client said something."""
    return len(client_says(user, "Knock! Knock!", "Bye.")) > 0


ONE_JOKE = ("Knock! Knock!", "Lettuce", "Lettuce in, it's cold out here! Want another?
(y/n)", "Bye.")
EXPECTED = ["Who's there?", "Lettuce who?", "n"]
```

Listing 16.12 (Continued) *protocol_contract_test.py: the cross tests in Python, one test class per member.*

```
class ClientMemberTestsTheServer(unittest.TestCase):
    def test_server_speaks_first(self):
        self.assertEqual("Knock! Knock!", server_says("Who's there?")[0])

    def test_one_whole_joke(self):
        self.assertEqual(["Knock! Knock!", "Lettuce",
                          "Lettuce in, it's cold out here! Want another? (y/n)", "Bye.",
"(finished)"],
                         server_says("Who's there?", "Lettuce who?", "n"))

    def test_case_does_not_matter(self):
        self.assertEqual("Lettuce", server_says("WHO'S THERE?")[1])

    def test_client_vanishes(self):
        self.assertEqual("(client left)", server_says("Who's there?")[2])


class ServerMemberTestsTheClient(unittest.TestCase):
    def test_client_follows_the_protocol(self):
        self.assertEqual(EXPECTED, client_says(PoliteUser(1), *ONE_JOKE))

    def test_client_stops_at_bye(self):
        self.assertEqual(["Who's there?"],
                         client_says(PoliteUser(1), "Knock! Knock!", "Bye.", "Knock!
Knock!"))

    def test_client_survives_a_hang_up(self):
        self.assertEqual(["Who's there?"], client_says(PoliteUser(1), "Knock! Knock!"))

    def test_strong_test_catches_the_sloppy_client(self):
        self.assertNotEqual(EXPECTED, client_says(SloppyUser(1), *ONE_JOKE))


if __name__ == "__main__":
    print("weak test passes the sloppy client:", weak_test(SloppyUser(1)))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
weak test passes the sloppy client: True
test_case_does_not_matter (__main__.ClientMemberTestsTheServer.test_case_does_not_matter)
... ok
test_client_vanishes (__main__.ClientMemberTestsTheServer.test_client_vanishes) ... ok
test_one_whole_joke (__main__.ClientMemberTestsTheServer.test_one_whole_joke) ... ok
test_server_speaks_first (__main__.ClientMemberTestsTheServer.test_server_speaks_first) ...
ok
test_client_follows_the_protocol
(__main__.ServerMemberTestsTheClient.test_client_follows_the_protocol) ... ok
test_client_stops_at_bye (__main__.ServerMemberTestsTheClient.test_client_stops_at_bye) ...
ok
test_client_survives_a_hang_up
(__main__.ServerMemberTestsTheClient.test_client_survives_a_hang_up) ... ok
test_strong_test_catches_the_sloppy_client
(__main__.ServerMemberTestsTheClient.test_strong_test_catches_the_sloppy_client) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.000s

OK
```

Line by line. `replay` joins the lines with newlines, adds a final newline, and wraps the text in a `StringReader` inside a `BufferedReader`; to the code under test this is indistinguishable from a socket's input stream. `serverSays` runs the real `KnockKnockServer.converse` with the stub as input and returns the server's lines plus a marker, `(finished)` or `(client left)`, taken from

`converse`'s return value. The client member's four tests check that the server speaks first, a whole joke, that case does not matter, and that a client who stops after one line is noticed. `clientSays` does the opposite for the server member: the stub server "says" fixed lines, and the test checks what the client wrote. The server member's three tests cover the normal joke, a client that must stop at `Bye.` even if the server says more, and a server that hangs up without saying `Bye.`. The last two are exactly the cases that the person who wrote the client is least likely to think of, which is the argument for letting the other side write the tests.

A tempting wrong solution. `weakTest` checks only that the client said *something*. It is the kind of test written in a hurry so that a report can say "tested", and it passes the mutant `SloppyUser`, which says "Who is there?", a reply the real server rejects. The strong test compares the client's lines with the exact expected lines and fails the mutant. Deliberately breaking the code and checking that some test fails is called *mutation testing*; it is the quickest way to find out whether your tests test anything [2].

> **CHECK YOURSELF.** Write the expected result of `serverSays("Who's there?", "Cabbage who?")` without running it.
>
> *Answer.* `[Knock! Knock!, Lettuce, You're supposed to say "Lettuce who?"! Try again. Knock! Knock!, (client left)]`. The server greets, gives the clue, rejects the wrong clue and knocks again, and then the replayed input ends, so `converse` returns `false`.

16.5.3 Problem 16.5 — Many clients at once: a multithreaded, distributed test

> **PROBLEM.** Develop and use a test plan that includes multithreaded and distributed testing. Start many clients at the same moment against one running server and check that every client receives exactly the conversation it should, and that the server counted every conversation. Show a server design that passes the single-client tests but fails this one.

In our own words. The tests of Problem 16.4 check each side alone. Now we check the running system under the condition it was built for: many clients talking at once over real sockets. Two questions make such a test trustworthy. What is the right answer for each client? And how do we make the clients really overlap instead of running one after another?

Design. The right answer comes from an *oracle*, a second, independent way of computing the expected result: for a client that wants n jokes, we run a fresh protocol and a `PoliteUser` together *without* a network and record the transcript. The overlap comes from a *start gate*: every client thread first waits on a `CountDownLatch` of 1 (Python: a `threading.Event`), and the test opens the gate only when all twenty tasks have been submitted, so they start within microseconds of each other. Every client also has a time limit and a limit of 30 turns, because a broken server must produce a failing test, not a test that hangs forever. Finally, the server updates its counters a moment *after* the client has read `Bye.`, so the test waits for the count with a deadline of two seconds instead of checking it immediately.

The Java solution. Listing 16.13 runs the test against the correct server and against a tempting "optimization": a server that creates one protocol object and shares it among all connections.

Listing 16.13 `ConcurrentClientsTest.java`: twenty clients released together by a start gate, checked against an oracle, and a server that shares one protocol object.

```
// ConcurrentClientsTest.java - Problem 16.5: many clients at once, and a server that
shares too much.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.function.Function;

public class ConcurrentClientsTest {

    static final int CLIENTS = 20;

    /** The oracle: the transcript a client wanting n jokes MUST see, computed without any
network. */
    static List<String> expected(int n) {
        KnockKnockProtocol p = new KnockKnockProtocol(JokeStore.classics());
        Function<String, String> user = new KnockKnockClient.PoliteUser(n);
        List<String> t = new ArrayList<>();
        String s = p.processInput(null);
        while (true) {
            t.add("S: " + s);
            if (s.equals("Bye.")) {
                return t;
            }
            String c = user.apply(s);
            t.add("C: " + c);
            s = p.processInput(c);
        }
    }
```

Listing 16.13 (Continued) *ConcurrentClientsTest.java: twenty clients released together by a start gate, checked against an oracle, and a server that shares one protocol object.*

```
    /** A client with a time limit and limited patience, so a broken server cannot hang the
test. */
    static List<String> talkWithin(int port, int n) throws IOException {
        Function<String, String> polite = new KnockKnockClient.PoliteUser(n);
        int[] turns = {0};
        Function<String, String> patient = line -> (++turns[0] > 30) ? null :
polite.apply(line);
        try (Socket s = new Socket("localhost", port);
             BufferedReader in = new BufferedReader(new InputStreamReader(
                     s.getInputStream(), StandardCharsets.UTF_8));
             PrintWriter out = new PrintWriter(new OutputStreamWriter(
                     s.getOutputStream(), StandardCharsets.UTF_8), true)) {
            s.setSoTimeout(2000);
            return KnockKnockClient.converse(in, out, patient);
        } catch (IOException e) {
            return List.of("(" + e.getClass().getSimpleName() + ")");
        }
    }

    /** Starts CLIENTS clients at the same instant; returns how many saw the right
transcript. */
    static int correctConversations(int port) throws Exception {
        ExecutorService pool = Executors.newFixedThreadPool(CLIENTS);
        CountDownLatch startGate = new CountDownLatch(1);
        List<Future<Boolean>> results = new ArrayList<>();
        for (int i = 0; i < CLIENTS; i++) {
            int n = i % 3 + 1;                              // 1, 2 or 3 jokes
            results.add(pool.submit(() -> {
                startGate.await();                          // everybody waits here ...
                return talkWithin(port, n).equals(expected(n));
            }));
        }
        startGate.countDown();                              // ... and starts together
        int correct = 0;
        for (Future<Boolean> f : results) {
            if (f.get()) {
                correct++;
            }
        }
        pool.shutdown();
        return correct;
    }
```

Listing 16.13 (Continued) *ConcurrentClientsTest.java: twenty clients released together by a start gate, checked against an oracle, and a server that shares one protocol object.*

```
    /** Tempting but wrong: ONE protocol object shared by every connection. */
    static int startSharedProtocolServer() throws IOException {
        ServerSocket listener = new ServerSocket(0);
        KnockKnockProtocol shared = new KnockKnockProtocol(JokeStore.classics());
        Thread acceptor = new Thread(() -> {
            try {
                while (true) {
                    Socket c = listener.accept();
                    new Thread(() -> {
                        try (c;
                             BufferedReader in = new BufferedReader(new
InputStreamReader(c.getInputStream()));
                             PrintWriter out = new PrintWriter(c.getOutputStream(), true))
{
                            KnockKnockServer.converse(in, out, shared);
                        } catch (IOException | RuntimeException e) {
                            // the conversation breaks; the socket is closed by try
                        }
                    }).start();
                }
            } catch (IOException e) {
                // listener closed
            }
        });
        acceptor.setDaemon(true);
        acceptor.start();
        return listener.getLocalPort();
    }

    static String firstLine(Socket s) throws IOException {
        return new BufferedReader(new InputStreamReader(s.getInputStream())).readLine();
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }
```

Listing 16.13 (Continued) *ConcurrentClientsTest.java: twenty clients released together by a start gate, checked against an oracle, and a server that shares one protocol object.*

```
    public static void main(String[] args) throws Exception {
        KnockKnockServer server = new KnockKnockServer(JokeStore.classics());
        int port = server.start(0);
        int good = correctConversations(port);
        System.out.println("correct server: " + good + " of " + CLIENTS + " conversations
right");
        long deadline = System.currentTimeMillis() + 2000;  // the server counts a moment
later
        while (server.finishedCount() < CLIENTS && System.currentTimeMillis() < deadline) {
            Thread.sleep(10);
        }
        check("every client got its own conversation", CLIENTS, good);
        check("server counted every conversation", CLIENTS, server.finishedCount());
        server.stop();

        int sharedPort = startSharedProtocolServer();
        try (Socket a = new Socket("localhost", sharedPort)) {
            System.out.println("shared protocol, client A hears: " + firstLine(a));
            try (Socket b = new Socket("localhost", sharedPort)) {     // B arrives after A
                System.out.println("shared protocol, client B hears: " + firstLine(b));
            }
        }
        int bad = correctConversations(sharedPort);
        System.out.println("shared protocol: all " + CLIENTS + " right? " + (bad ==
CLIENTS));
        check("sharing the protocol breaks conversations", true, bad < CLIENTS);
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
correct server: 20 of 20 conversations right
PASS every client got its own conversation
PASS server counted every conversation
shared protocol, client A hears: Knock! Knock!
shared protocol, client B hears: You're supposed to say "Who's there?"! Try again. Knock!
Knock!
shared protocol: all 20 right? false
PASS sharing the protocol breaks conversations
All 3 checks passed.
```

The Python solution. **Listing 16.14** uses a `ThreadPoolExecutor` for the clients and `threading.Event` as the start gate.

Listing 16.14 `concurrent_clients_test.py`: the same distributed test in Python.

```
# concurrent_clients_test.py - Problem 16.5: many clients at once, and a server that shares
too much.
import socket
import sys
import threading
import time
import unittest
from concurrent.futures import ThreadPoolExecutor
from joke_store import classics
from knock_knock_protocol import KnockKnockProtocol
from knock_knock_server import KnockKnockServer, converse as server_converse
from knock_knock_client import PoliteUser, converse as client_converse

CLIENTS = 20


def expected(n):
    """The oracle: the transcript a client wanting n jokes MUST see, computed without any
network."""
    p, user, t = KnockKnockProtocol(classics()), PoliteUser(n), []
    s = p.process_input(None)
    while True:
        t.append("S: " + s)
        if s == "Bye.":
            return t
        c = user(s)
        t.append("C: " + c)
        s = p.process_input(c)


def talk_within(port, n):
    """A client with a time limit and limited patience, so a broken server cannot hang the
test."""
    polite, turns = PoliteUser(n), [0]

    def patient(line):
        turns[0] += 1
        return None if turns[0] > 30 else polite(line)

    try:
        with socket.create_connection(("localhost", port), timeout=2) as s, \
                s.makefile("r", encoding="utf-8") as reader, \
                s.makefile("w", encoding="utf-8") as writer:
            return client_converse(reader, writer, patient)
    except OSError as e:
        return [f"({type(e).__name__})"]
```

Listing 16.14 (Continued) *concurrent_clients_test.py: the same distributed test in Python.*

```
def correct_conversations(port):
    """Starts CLIENTS clients at the same instant; returns how many saw the right
transcript."""
    start_gate = threading.Event()

    def one_client(n):
        start_gate.wait()                               # everybody waits here ...
        return talk_within(port, n) == expected(n)

    with ThreadPoolExecutor(max_workers=CLIENTS) as pool:
        futures = [pool.submit(one_client, i % 3 + 1) for i in range(CLIENTS)]
        start_gate.set()                                # ... and starts together
        return sum(1 for f in futures if f.result())


def start_shared_protocol_server():
    """Tempting but wrong: ONE protocol object shared by every connection."""
    listener = socket.create_server(("localhost", 0))
    shared = KnockKnockProtocol(classics())

    def serve(c):
        try:
            with c, c.makefile("r") as reader, c.makefile("w") as writer:
                server_converse(reader, writer, shared)
        except (OSError, RuntimeError):
            pass                                        # the conversation breaks

    def accept_loop():
        while True:
            c, _ = listener.accept()
            threading.Thread(target=serve, args=(c,), daemon=True).start()

    threading.Thread(target=accept_loop, daemon=True).start()
    return listener.getsockname()[1]


def first_line(s):
    return s.makefile("r").readline().rstrip("\n")
```

Listing 16.14 (Continued) *concurrent_clients_test.py: the same distributed test in Python.*

```
class TestConcurrentClients(unittest.TestCase):
    def test_every_client_gets_its_own_conversation(self):
        server = KnockKnockServer(classics())
        good = correct_conversations(server.start(0))
        deadline = time.monotonic() + 2                 # the server counts a moment later
        while server.counts()[0] < CLIENTS and time.monotonic() < deadline:
            time.sleep(0.01)
        server.stop()
        self.assertEqual(CLIENTS, good)
        self.assertEqual(CLIENTS, server.counts()[0])

    def test_sharing_the_protocol_breaks_conversations(self):
        self.assertLess(correct_conversations(start_shared_protocol_server()), CLIENTS)


if __name__ == "__main__":
    port = start_shared_protocol_server()
    with socket.create_connection(("localhost", port)) as a:
        print("shared protocol, client A hears:", first_line(a))
        with socket.create_connection(("localhost", port)) as b:     # B arrives after A
            print("shared protocol, client B hears:", first_line(b))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
shared protocol, client A hears: Knock! Knock!
shared protocol, client B hears: You're supposed to say "Who's there?"! Try again. Knock!
Knock!
test_every_client_gets_its_own_conversation
(__main__.TestConcurrentClients.test_every_client_gets_its_own_conversation) ... ok
test_sharing_the_protocol_breaks_conversations
(__main__.TestConcurrentClients.test_sharing_the_protocol_breaks_conversations) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.059s

OK
```

Line by line. `expected(n)` is the oracle: it alternates `processInput` and `PoliteUser` until `Bye.`. `talkWithin` wraps the user in a second function, `patient`, that counts turns in a one-element array (a lambda may not assign to an ordinary local variable, but it may change an array element) and gives up after 30. `correctConversations` submits twenty tasks to a fixed pool; each calls `startGate.await()` and waits there. The call `startGate.countDown()` lowers the count from 1 to 0 and releases all of them together. Each `Future<Boolean>` holds one client's verdict, and `f.get()` waits for it. The loop after `correctConversations` in `main` polls `finishedCount()` every 10 ms for at most two seconds: this is how to test something that becomes true *eventually*, a common situation in distributed systems.

The wrong server is ordinary except for one line: `KnockKnockProtocol shared = new KnockKnockProtocol(...)` sits *outside* the accept loop. The demonstration before the stress test is fully deterministic. Client A connects and reads its first line: the shared protocol moves from `WAITING` to `SENT_KNOCK` and A hears "Knock! Knock!". Only then does client B connect; the server calls `processInput(null)` on the same object, which is now in `SENT_KNOCK`, and B's very first line is "You're supposed to say …". Under the stress test the conversations interleave arbitrarily, and not all twenty are right. The Python program repeats the demonstration before running its tests.

A tempting wrong solution. Sharing the protocol object feels like saving memory, and a test with one client cannot tell the difference. The bug is not really about threads: the two clients of the demonstration are served one after the other and still interfere, because a conversation's state

lives in an object that two conversations use. The rule from Chapter 14 is worth restating in this form: *per-connection state belongs to the connection*; what is shared must be immutable (the joke store) or protected (the counters).

> **CHECK YOURSELF.** Someone proposes to fix the shared-protocol server by declaring `processInput` `synchronized`. Would the stress test pass?
>
> *Answer.* No. `synchronized` makes each call run alone, so the fields can no longer be corrupted halfway through a call, but the calls of different clients still interleave between calls: A's knock followed by B's greeting is exactly the deterministic failure, and it involves no simultaneous calls at all. Thread-safety of each method does not make a shared conversation correct; only one protocol object per connection does.

16.6 Problems on Robustness, the Interface, and Documentation

16.6.1 Problem 16.6 — When the server or a client goes down

> **PROBLEM.** Take care of failures on both sides. When the server is down, the client should know it; when a client is down, the server should know it. Handle connection errors as exceptions, both when the server is down and when a client is down, and show that the server keeps serving other clients after one of them has vanished.

In our own words. A network program meets failures that a single program never sees, and each one looks different from the inside. We will list them, decide what each side observes in each case, and write a client whose only way of ending is a clear sentence for the user, never a stack trace.

Design. **Table 16.6** is the design. Its middle columns are the part that needs knowledge of sockets: what the program actually *observes*, which is different in Java and in Python.

Table 16.6 What each side observes when the other side fails, in Java and in Python, and how the Knock-Knock Club reacts.

Situation	Java observes	Python observes	Our reaction
server not running	`ConnectException` from `new Socket` or `connect`	`ConnectionRefusedError`	"Cannot reach the server: is it running?"
server hangs up mid-conversation	`readLine()` returns `null`	`readline()` returns `""`	"The server closed the connection after *k* line(s)."
server alive but silent	`SocketTimeoutException` after `setSoTimeout`	`TimeoutError` after `settimeout`	"The server did not answer within 500 ms."
client leaves normally or crashes	server's `readLine()` returns `null`	server's `readline()` returns `""`	log "client left before the end", keep serving
connection reset (unread data, cable pulled)	`SocketException: Connection reset`	`ConnectionResetError`	log "connection lost", keep serving

Two design rules follow. First, *every blocking call gets a time limit*: `connect` with a timeout and `setSoTimeout` for reads in Java, `create_connection(..., timeout=...)` in Python. Without a limit, a silent server freezes the client forever, and no exception is ever thrown to be handled. Second, *end of stream is a normal result, not an error*: `null` in Java and `""` in Python must be tested for on every read.

The Java solution. **Listing 16.15** contains the safe client, a naive client, two helpers that create failures on purpose (a port where nobody listens, and a misbehaving server), and a scenario in which a client leaves in the middle and a second client is served normally.

Listing 16.15 `FailureDetection.java`: a client that turns every failure into one sentence, a naive client, deliberately misbehaving servers, and a server that notices a departed client.

```
// FailureDetection.java - Problem 16.6: the client notices a dead server, the server
notices a dead client.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ConnectException;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.nio.charset.StandardCharsets;
import java.util.List;
import java.util.function.Function;

public class FailureDetection {

    /** The client's safe entry point: every network failure becomes one sentence for the
user. */
    static String safeTalk(int port, int limitMillis, Function<String, String> user) {
        try (Socket s = new Socket()) {
            s.connect(new InetSocketAddress("localhost", port), limitMillis);  // connect,
with a limit
            s.setSoTimeout(limitMillis);                                       // every
read, with a limit
            BufferedReader in = new BufferedReader(new InputStreamReader(
                    s.getInputStream(), StandardCharsets.UTF_8));
            PrintWriter out = new PrintWriter(new OutputStreamWriter(
                    s.getOutputStream(), StandardCharsets.UTF_8), true);
            List<String> t = KnockKnockClient.converse(in, out, user);
            if (t.isEmpty() || !t.get(t.size() - 1).equals("S: Bye.")) {
                return "The server closed the connection after " + t.size() + " line(s).";
            }
            return "OK, " + t.size() + " lines.";
        } catch (ConnectException e) {
            return "Cannot reach the server: is it running?";
        } catch (SocketTimeoutException e) {
            return "The server did not answer within " + limitMillis + " ms.";
        } catch (IOException e) {
            return "Connection error: " + e.getClass().getSimpleName();
        }
    }
```

Listing 16.15 (Continued) *FailureDetection.java: a client that turns every failure into one sentence, a naive client, deliberately misbehaving servers, and a server that notices a departed client.*

```
    /** Tempting but wrong: a client that assumes the server is always there. */
    static void naiveTalk(int port) throws IOException {
        Socket s = new Socket("localhost", port);
        BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));
        PrintWriter out = new PrintWriter(s.getOutputStream(), true);
        Function<String, String> user = new KnockKnockClient.PoliteUser(1);
        while (true) {
            String line = in.readLine();
            if (line.equals("Bye.")) {                    // line is null after a hang-up
                break;
            }
            out.println(user.apply(line));
        }
        s.close();
    }

    /** A port on which nobody listens: we open it, note its number and close it again. */
    static int portWithNobody() throws IOException {
        try (ServerSocket probe = new ServerSocket(0)) {
            return probe.getLocalPort();
        }
    }

    /** A misbehaving server: says its lines, hears one reply, then hangs up (or stays
silent). */
    static int badServer(List<String> lines, boolean hangUp) throws IOException {
        ServerSocket listener = new ServerSocket(0);
        Thread t = new Thread(() -> {
            try (listener; Socket c = listener.accept()) {
                PrintWriter out = new PrintWriter(c.getOutputStream(), true);
                lines.forEach(out::println);
                if (hangUp) {
                    new BufferedReader(new
InputStreamReader(c.getInputStream())).readLine();
                } else {
                    Thread.sleep(5000);                   // alive, but silent
                }
            } catch (IOException | InterruptedException e) {
                // nothing to do: this server is meant to misbehave
            }
        });
        t.setDaemon(true);
        t.start();
        return listener.getLocalPort();
    }

    private static int passed = 0;
```

Listing 16.15 (Continued) *FailureDetection.java: a client that turns every failure into one sentence, a naive client, deliberately misbehaving servers, and a server that notices a departed client.*

```
    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) throws Exception {
        String down = safeTalk(portWithNobody(), 500, new KnockKnockClient.PoliteUser(1));
        String crash = safeTalk(badServer(List.of("Knock! Knock!"), true), 500,
                                new KnockKnockClient.PoliteUser(1));
        String silent = safeTalk(badServer(List.of(), false), 500, new
KnockKnockClient.PoliteUser(1));
        System.out.println("server down    -> " + down);
        System.out.println("server crashes -> " + crash);
        System.out.println("server silent  -> " + silent);
        try {
            naiveTalk(badServer(List.of("Knock! Knock!"), true));
        } catch (NullPointerException e) {
            System.out.println("naive client   -> NullPointerException");
        }

        KnockKnockServer server = new KnockKnockServer(JokeStore.classics());
        int port = server.start(0);
        try (Socket quitter = new Socket("localhost", port)) {
            new BufferedReader(new InputStreamReader(quitter.getInputStream())).readLine();
        }                                                    // leaves after the first line
        String next = safeTalk(port, 500, new KnockKnockClient.PoliteUser(1));
        long deadline = System.currentTimeMillis() + 2000;
        while (server.finishedCount() + server.leftEarlyCount() < 2
                && System.currentTimeMillis() < deadline) {
            Thread.sleep(10);
        }
        System.out.println("next client    -> " + next);
        System.out.println("server log     -> " + server.log());
        server.stop();

        check("server down", "Cannot reach the server: is it running?", down);
        check("server crashes", "The server closed the connection after 2 line(s).",
crash);
        check("server silent", "The server did not answer within 500 ms.", silent);
        check("server survives a client that leaves", "OK, 7 lines.", next);
        check("server noticed", List.of("client left before the end"), server.log());
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
server down    -> Cannot reach the server: is it running?
server crashes -> The server closed the connection after 2 line(s).
server silent  -> The server did not answer within 500 ms.
naive client   -> NullPointerException
next client    -> OK, 7 lines.
server log     -> [client left before the end]
PASS server down
PASS server crashes
PASS server silent
PASS server survives a client that leaves
PASS server noticed
All 5 checks passed.
```

The Python solution. **Listing 16.16** follows the same plan.

Listing 16.16 `failure_detection.py`: the same failure scenarios in Python.

```
# failure_detection.py - Problem 16.6: the client notices a dead server, the server notices
a dead client.
import socket
import sys
import threading
import time
import unittest
from joke_store import classics
from knock_knock_server import KnockKnockServer, send
from knock_knock_client import PoliteUser, converse


def safe_talk(port, limit_seconds, user):
    """The client's safe entry point: every network failure becomes one sentence for the
user."""
    try:
        with socket.create_connection(("localhost", port), timeout=limit_seconds) as s, \
                s.makefile("r", encoding="utf-8") as reader, \
                s.makefile("w", encoding="utf-8") as writer:
            t = converse(reader, writer, user)          # the timeout covers every read
    except ConnectionRefusedError:
        return "Cannot reach the server: is it running?"
    except TimeoutError:
        return f"The server did not answer within {int(limit_seconds * 1000)} ms."
    except OSError as e:
        return f"Connection error: {type(e).__name__}"
    if not t or t[-1] != "S: Bye.":
        return f"The server closed the connection after {len(t)} line(s)."
    return f"OK, {len(t)} lines."


def naive_talk(port):
    """Tempting but wrong: a client that assumes the server is always there."""
    s = socket.create_connection(("localhost", port))
    reader, writer, user = s.makefile("r"), s.makefile("w"), PoliteUser(1)
    while True:
        line = reader.readline().rstrip("\n")           # "" after a hang-up, forever
        if line == "Bye.":
            break
        send(writer, user(line))                        # answers " who?" to silence


def port_with_nobody():
    """A port on which nobody listens: we open it, note its number and close it again."""
    with socket.create_server(("localhost", 0)) as probe:
        return probe.getsockname()[1]
```

Listing 16.16 (Continued) *failure_detection.py: the same failure scenarios in Python.*

```
def bad_server(lines, hang_up):
    """A misbehaving server: says its lines, hears one reply, then hangs up (or stays
silent)."""
    listener = socket.create_server(("localhost", 0))

    def run():
        with listener, listener.accept()[0] as c, c.makefile("r") as reader,
c.makefile("w") as writer:
            for line in lines:
                send(writer, line)
            if hang_up:
                reader.readline()
            else:
                time.sleep(5)                           # alive, but silent

    threading.Thread(target=run, daemon=True).start()
    return listener.getsockname()[1]


def client_down_scenario():
    """A client leaves after one line; a second client must still be served."""
    server = KnockKnockServer(classics())
    port = server.start(0)
    with socket.create_connection(("localhost", port)) as quitter:
        quitter.makefile("r").readline()                # leaves after the first line
    nxt = safe_talk(port, 0.5, PoliteUser(1))
    deadline = time.monotonic() + 2
    while sum(server.counts()[:2]) < 2 and time.monotonic() < deadline:
        time.sleep(0.01)
    server.stop()
    return nxt, server.counts()[2]


class TestFailures(unittest.TestCase):
    def test_server_down(self):
        self.assertEqual("Cannot reach the server: is it running?",
                         safe_talk(port_with_nobody(), 0.5, PoliteUser(1)))

    def test_server_crashes(self):
        self.assertEqual("The server closed the connection after 2 line(s).",
                         safe_talk(bad_server(["Knock! Knock!"], True), 0.5,
PoliteUser(1)))

    def test_server_silent(self):
        self.assertEqual("The server did not answer within 500 ms.",
                         safe_talk(bad_server([], False), 0.5, PoliteUser(1)))
```

Listing 16.16 (Continued) *failure_detection.py: the same failure scenarios in Python.*

```
    def test_naive_client_crashes(self):
        with self.assertRaises(OSError):                # BrokenPipeError or
ConnectionResetError
            naive_talk(bad_server(["Knock! Knock!"], True))

    def test_server_survives_a_client_that_leaves(self):
        nxt, log = client_down_scenario()
        self.assertEqual("OK, 7 lines.", nxt)
        self.assertEqual(["client left before the end"], log)


if __name__ == "__main__":
    print("server down    ->", safe_talk(port_with_nobody(), 0.5, PoliteUser(1)))
    print("server crashes ->", safe_talk(bad_server(["Knock! Knock!"], True), 0.5,
PoliteUser(1)))
    print("server silent  ->", safe_talk(bad_server([], False), 0.5, PoliteUser(1)))
    try:
        naive_talk(bad_server(["Knock! Knock!"], True))
    except OSError as e:
        print("naive client   -> crashed with an OSError:", isinstance(e, OSError))
    nxt, log = client_down_scenario()
    print("next client    ->", nxt)
    print("server log     ->", log)
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
server down    -> Cannot reach the server: is it running?
server crashes -> The server closed the connection after 2 line(s).
server silent  -> The server did not answer within 500 ms.
naive client   -> crashed with an OSError: True
next client    -> OK, 7 lines.
server log     -> ['client left before the end']
test_naive_client_crashes (__main__.TestFailures.test_naive_client_crashes) ...
/usr/lib/python3.11/socket.py:789: ResourceWarning: unclosed <socket.socket fd=7, family=2,
type=1, proto=6, laddr=('127.0.0.1', 46800)>
  self._sock = None
ResourceWarning: Enable tracemalloc to get the object allocation traceback
ok
test_server_crashes (__main__.TestFailures.test_server_crashes) ... ok
test_server_down (__main__.TestFailures.test_server_down) ... ok
test_server_silent (__main__.TestFailures.test_server_silent) ... ok
test_server_survives_a_client_that_leaves
(__main__.TestFailures.test_server_survives_a_client_that_leaves) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.521s

OK
```

Line by line. `safeTalk` creates an unconnected `Socket` and calls `connect(address, limitMillis)`, so even the connection attempt has a time limit (a plain `new Socket(host, port)` may wait more than a minute for an unreachable machine). `setSoTimeout(limitMillis)` then limits every read. The three `catch` clauses go from specific to general, which Java requires: `ConnectException` and `SocketTimeoutException` are both subclasses of `IOException`, and a general clause written first would make the specific ones unreachable, a compile-time error. After a normal return, the transcript itself tells whether the conversation ended properly: if its last line is not `S: Bye.`, the server hung up early. `portWithNobody` opens a server socket on a free port, remembers the number, and closes it again: a connection attempt to that port is refused. `badServer` accepts one client, says its lines, and then either reads one reply and hangs up or stays

silent for five seconds. Reading one reply before hanging up makes the scenario deterministic: the server closes only after the client's line has arrived, so the client sees a clean end of stream.

The last scenario uses the real server. `quitter` reads one line and is closed by its `try` block, which sends the server an end-of-stream; the worker's `readLine()` returns `null`, `converse` returns `false`, and the server logs "client left before the end". A second client, started after that, completes a normal one-joke conversation (7 lines), which shows that one departed client does not disturb the others. The server's log is checked for exactly one entry.

In Python, `socket.create_connection((host, port), timeout=0.5)` applies the time limit to the connection *and* to every later operation on the socket. The exception classes differ, `ConnectionRefusedError` and `TimeoutError`, but the structure of `safe_talk` is the same.

A tempting wrong solution. `naiveTalk` is the client most people write first: `while (true) { String line = in.readLine(); if (line.equals("Bye.")) break; ... }`. When the server vanishes, `readLine()` returns `null`, and `line.equals(...)` throws a `NullPointerException`, which the output shows. The Python version fails differently, and more dangerously: `readline()` returns `""` forever after the end of the stream, so the loop keeps answering " who?" to silence until writing to the dead socket raises a `BrokenPipeError` or `ConnectionResetError`. Either way the user sees a crash instead of a sentence, and in both languages the naive client also leaves its socket open, which `unittest` even reports in the Python output as a `ResourceWarning`. Test for the end of the stream on every read, and close sockets in `try`-with-resources or `with`.

> **CHECK YOURSELF.** A client's network cable is unplugged in the middle of a joke. The client's computer sends nothing, not even an end-of-stream. What does the server of **Listing 16.7** observe, and how would you make it notice?
>
> *Answer.* Nothing at all: the worker waits in `readLine()` forever, because no packet arrives to say that the connection is gone. Give the server's sockets a read timeout too (for example `client.setSoTimeout(60_000)` in `serve`), and treat a `SocketTimeoutException` as "client lost". A protocol that must detect silent failures quickly adds a *heartbeat*, a small message such as `PING` that each side sends every few seconds; missing heartbeats mean the other side is gone. Exercise 16.7 asks for this design.

16.6.2 Problem 16.7 — A window that never freezes

> **PROBLEM.** Give the client a graphical user interface. The window shows the conversation, lets the user type and send each reply, and shows the connection status, including "server down". The window must stay responsive while the program waits for the network, and the logic must be testable without opening a window.

In our own words. Chapter 10 taught one rule about graphical interfaces that becomes critical with networks: the thread that draws the window and handles clicks (Swing's *event dispatch thread*, Tkinter's main loop) must never wait for anything. A network read can wait for seconds. So the network must live on its own thread, and the two threads must talk to each other safely. Chapter 11 gives us the structure: a *view* that only displays, and a *controller* that does the work.

Design. The controller runs the whole conversation on a background thread called `network`. It needs three things, and each is passed in as an object, which is what makes it testable. A `View` (an interface with `showLine`, `setStatus` and `setInputEnabled`) is where it shows things. An `Executor` called `ui` is *how* it shows them: every change to the view is wrapped in a small `Runnable` and handed to `ui`, which in the real program is `SwingUtilities::invokeLater`, the method that runs code on the event dispatch thread. Finally, a `Function<String, String>` supplies the user's

replies: in the window it waits on a `BlockingQueue` that the **Send** button fills; in the test it is `PoliteUser`. In the self-test, `ui` is simply `Runnable::run` (run at once) and the view is a `RecordingView` that stores every call in a list, so the test can check exactly what the user would have seen.

Figure 16.6 is a real screenshot of the Swing window of **Listing 16.17** in the middle of a conversation driven by the controller: the conversation area, the input field with its **Send** button, and the status line at the bottom. The Tkinter window of **Listing 16.18** has the same three parts.

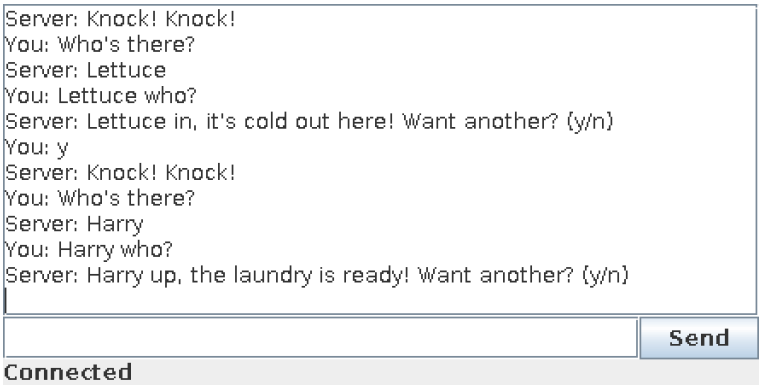


Figure 16.6 The Knock-Knock Club window (Swing) during a conversation. The status line reads "Connected"; the input field is enabled because the server is waiting for an answer.

The Java solution. **Listing 16.17** contains the interface, the controller, the Swing view, the recording view and a measurement that shows why the network must not run on the user-interface thread. Run it with `java KnockKnockGui` for the window (with the server already running), or with `java KnockKnockGui --selftest` for the headless test.

Listing 16.17 `KnockKnockGui.java`: MVC with a `View` interface, a controller that runs the network on its own thread, a Swing view, and a headless self-test.

```
// KnockKnockGui.java - Problem 16.7: a Swing client that stays responsive, with a headless
self-test.
import java.awt.BorderLayout;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ConnectException;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;
import java.util.function.Function;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class KnockKnockGui {

    /** What the controller needs from any view: the V of MVC, as an interface. */
    interface View {
        void showLine(String line);

        void setStatus(String status);

        void setInputEnabled(boolean enabled);
    }

    /** The controller: talks to the server on a background thread, never on the UI thread.
*/
    static final class Controller {
        private final View view;
        private final Executor ui;                          // runs code on the UI thread
        private final Function<String, String> replies;     // where the user's answers
come from
```

Listing 16.17 (Continued) *KnockKnockGui.java: MVC with a View interface, a controller that runs the network on its own thread, a Swing view, and a headless self-test.*

```
        Controller(View view, Executor ui, Function<String, String> replies) {
            this.view = view;
            this.ui = ui;
            this.replies = replies;
        }

        /** Starts the conversation and returns at once; the network thread does the
waiting. */
        Thread connect(int port) {
            Thread network = new Thread(() -> talk(port), "network");
            network.setDaemon(true);
            network.start();
            return network;
        }

        private void talk(int port) {
            ui.execute(() -> view.setStatus("Connecting..."));
            try (Socket s = new Socket("localhost", port);
                 BufferedReader in = new BufferedReader(new InputStreamReader(
                         s.getInputStream(), StandardCharsets.UTF_8));
                 PrintWriter out = new PrintWriter(new OutputStreamWriter(
                         s.getOutputStream(), StandardCharsets.UTF_8), true)) {
                ui.execute(() -> view.setStatus("Connected"));
                List<String> t = KnockKnockClient.converse(in, out, line -> {
                    ui.execute(() -> {
                        view.showLine("Server: " + line);
                        view.setInputEnabled(true);
                    });
                    String reply = replies.apply(line);             // may wait for the
user
                    ui.execute(() -> {
                        view.setInputEnabled(false);
                        view.showLine("You: " + reply);
                    });
                    return reply;
                });
                String last = t.get(t.size() - 1);
                ui.execute(() -> {
                    if (last.equals("S: Bye.")) {
                        view.showLine("Server: Bye.");
                        view.setStatus("Conversation over");
                    } else {
                        view.setStatus("The server closed the connection");
                    }
                });
            } catch (ConnectException e) {
                ui.execute(() -> view.setStatus("Cannot reach the server: is it
running?"));
            } catch (IOException e) {
```

Listing 16.17 (Continued) *KnockKnockGui.java: MVC with a View interface, a controller that runs the network on its own thread, a Swing view, and a headless self-test.*

```
                ui.execute(() -> view.setStatus("Connection lost: " +
e.getClass().getSimpleName()));
            }
        }
    }

    /** The real window. Every method is called on the Swing thread (the EDT). */
    static final class SwingView implements View {
        private final JTextArea conversation = new JTextArea(12, 40);
        private final JTextField input = new JTextField(30);
        private final JButton send = new JButton("Send");
        private final JLabel status = new JLabel("Not connected");

        SwingView(BlockingQueue<String> answers) {
            conversation.setEditable(false);
            send.addActionListener(e -> {                   // runs on the EDT: must be
quick
                answers.add(input.getText());               // hand the answer to the
network thread
                input.setText("");
            });
            input.addActionListener(e -> send.doClick());   // Enter works like the button
            JPanel bottom = new JPanel(new BorderLayout());
            bottom.add(input, BorderLayout.CENTER);
            bottom.add(send, BorderLayout.EAST);
            bottom.add(status, BorderLayout.SOUTH);
            JFrame frame = new JFrame("Knock-Knock Club");
            frame.add(new JScrollPane(conversation), BorderLayout.CENTER);
            frame.add(bottom, BorderLayout.SOUTH);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.pack();
            frame.setVisible(true);
            setInputEnabled(false);
        }

        @Override
        public void showLine(String line) {
            conversation.append(line + "\n");
        }

        @Override
        public void setStatus(String text) {
            status.setText(text);
        }
```

Listing 16.17 (Continued) *KnockKnockGui.java: MVC with a View interface, a controller that runs the network on its own thread, a Swing view, and a headless self-test.*

```
        @Override
        public void setInputEnabled(boolean enabled) {
            input.setEnabled(enabled);
            send.setEnabled(enabled);
        }
    }

    /** A view for tests: it only remembers what it was asked to show. */
    static final class RecordingView implements View {
        final List<String> calls = Collections.synchronizedList(new ArrayList<>());

        @Override
        public void showLine(String line) {
            calls.add(line);
        }

        @Override
        public void setStatus(String status) {
            calls.add("[status] " + status);
        }

        @Override
        public void setInputEnabled(boolean enabled) {
            // not recorded: it would only make the test longer
        }
    }

    /** How long a "repaint" waits when the network call runs on the UI thread itself. */
    static long repaintDelay(boolean networkOnUiThread, int port) throws Exception {
        ExecutorService fakeEdt = Executors.newSingleThreadExecutor();
        Runnable blockingCall = () -> FailureDetection.safeTalk(port, 500,
                new KnockKnockClient.PoliteUser(1));        // waits 500 ms for a silent
server
        if (networkOnUiThread) {
            fakeEdt.execute(blockingCall);                  // wrong: the UI thread is busy
waiting
        } else {
            new Thread(blockingCall).start();               // right: a separate thread
waits
        }
        long clicked = System.nanoTime();
        fakeEdt.submit(() -> { }).get();                    // the repaint the user is
waiting for
        long waited = (System.nanoTime() - clicked) / 1_000_000;
        fakeEdt.shutdown();
        fakeEdt.awaitTermination(2, TimeUnit.SECONDS);
        return waited;
    }
```

Listing 16.17 (Continued) *KnockKnockGui.java: MVC with a View interface, a controller that runs the network on its own thread, a Swing view, and a headless self-test.*

```
    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) throws Exception {
        if (args.length > 0 && args[0].equals("--selftest")) {
            KnockKnockServer server = new KnockKnockServer(JokeStore.classics());
            int port = server.start(0);
            RecordingView view = new RecordingView();
            new Controller(view, Runnable::run, new
KnockKnockClient.PoliteUser(1)).connect(port).join();
            view.calls.forEach(System.out::println);
            server.stop();
            check("ten view calls", 10, view.calls.size());
            check("last status", "[status] Conversation over", view.calls.get(9));

            RecordingView lonely = new RecordingView();
            new Controller(lonely, Runnable::run, new KnockKnockClient.PoliteUser(1))
                    .connect(FailureDetection.portWithNobody()).join();
            check("server down is shown", "[status] Cannot reach the server: is it
running?",
                  lonely.calls.get(1));

            int silent = FailureDetection.badServer(List.of(), false);
            long wrong = repaintDelay(true, silent);
            long right = repaintDelay(false, FailureDetection.badServer(List.of(), false));
            System.out.println("repaint waited at least 400 ms, network on UI thread: " +
(wrong >= 400));
            System.out.println("repaint waited at least 400 ms, network on own thread: " +
(right >= 400));
            check("UI thread blocked by the wrong design", true, wrong >= 400);
            check("UI thread free in the right design", true, right < 400);
            System.out.println("All " + passed + " checks passed.");
            return;
        }
        int port = (args.length > 0) ? Integer.parseInt(args[0]) : 4444;
        BlockingQueue<String> answers = new LinkedBlockingQueue<>();
        SwingUtilities.invokeLater(() -> {
            View view = new SwingView(answers);
            Function<String, String> fromUser = line -> {
                try {
                    return answers.take();                  // the network thread waits
here
                } catch (InterruptedException e) {
                    return null;
```

Listing 16.17 (Continued) *KnockKnockGui.java: MVC with a View interface, a controller that runs the network on its own thread, a Swing view, and a headless self-test.*

```
                }
            };
            new Controller(view, SwingUtilities::invokeLater, fromUser).connect(port);
        });
    }
}
```

```
[status] Connecting...
[status] Connected
Server: Knock! Knock!
You: Who's there?
Server: Lettuce
You: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
You: n
Server: Bye.
[status] Conversation over
PASS ten view calls
PASS last status
PASS server down is shown
repaint waited at least 400 ms, network on UI thread: true
repaint waited at least 400 ms, network on own thread: false
PASS UI thread blocked by the wrong design
PASS UI thread free in the right design
All 5 checks passed.
```

The Python solution. **Listing 16.18** has the same parts. Tkinter is not thread-safe at all, so the network thread never calls the view directly: it puts small functions into a `queue.Queue`, and the Tk thread takes them out every 50 ms in `drain`. The module imports `tkinter` only inside the window code, so the self-test runs even on a machine without Tk.

Listing 16.18 `knock_knock_gui.py`: the same design with Tkinter; the network thread posts view updates into a queue that the Tk thread drains.

```
# knock_knock_gui.py - Problem 16.7: a Tkinter client that stays responsive, with a
headless self-test.
import queue
import socket
import sys
import threading
import time
import unittest
from concurrent.futures import ThreadPoolExecutor
from joke_store import classics
from knock_knock_server import KnockKnockServer
from knock_knock_client import PoliteUser, converse
from failure_detection import safe_talk, port_with_nobody, bad_server


class Controller:
    """Talks to the server on a background thread, never on the UI thread."""

    def __init__(self, view, ui, replies):
        self._view = view           # anything with show_line, set_status,
set_input_enabled
        self._ui = ui               # ui(f) runs the function f on the UI thread
        self._replies = replies     # where the user's answers come from

    def connect(self, port):
        """Starts the conversation and returns at once; the network thread does the
waiting."""
        network = threading.Thread(target=self._talk, args=(port,), daemon=True)
        network.start()
        return network

    def _answer(self, line):
        def before():
            self._view.show_line("Server: " + line)
            self._view.set_input_enabled(True)
        self._ui(before)
        reply = self._replies(line)                    # may wait for the user

        def after():
            self._view.set_input_enabled(False)
            self._view.show_line(f"You: {reply}")
        self._ui(after)
        return reply
```

Listing 16.18 (Continued) *knock_knock_gui.py: the same design with Tkinter; the network thread posts view updates into a queue that the Tk thread drains.*

```
    def _talk(self, port):
        self._ui(lambda: self._view.set_status("Connecting..."))
        try:
            with socket.create_connection(("localhost", port)) as s, \
                    s.makefile("r", encoding="utf-8") as reader, \
                    s.makefile("w", encoding="utf-8") as writer:
                self._ui(lambda: self._view.set_status("Connected"))
                t = converse(reader, writer, self._answer)
        except ConnectionRefusedError:
            self._ui(lambda: self._view.set_status("Cannot reach the server: is it
running?"))
            return
        except OSError as e:
            name = type(e).__name__
            self._ui(lambda: self._view.set_status("Connection lost: " + name))
            return

        def finish():
            if t and t[-1] == "S: Bye.":
                self._view.show_line("Server: Bye.")
                self._view.set_status("Conversation over")
            else:
                self._view.set_status("The server closed the connection")
        self._ui(finish)


class TkView:
    """The real window. Every method runs on the Tk thread."""

    def __init__(self, root, answers):
        import tkinter as tk                            # imported only when a window is
wanted
        root.title("Knock-Knock Club")
        self._conversation = tk.Text(root, width=48, height=12, state="disabled")
        self._conversation.pack(fill="both", expand=True)
        bottom = tk.Frame(root)
        bottom.pack(fill="x")
        self._input = tk.Entry(bottom, width=36)
        self._input.pack(side="left", fill="x", expand=True)
        self._send = tk.Button(bottom, text="Send", command=self._on_send)
        self._send.pack(side="right")
        self._status = tk.Label(root, text="Not connected", anchor="w")
        self._status.pack(fill="x")
        self._input.bind("<Return>", lambda event: self._on_send())
        self._answers = answers
        self.set_input_enabled(False)
```

Listing 16.18 (Continued) *knock_knock_gui.py: the same design with Tkinter; the network thread posts view updates into a queue that the Tk thread drains.*

```
    def _on_send(self):                                  # runs on the Tk thread: must be
quick
        self._answers.put(self._input.get())             # hand the answer to the network
thread
        self._input.delete(0, "end")

    def show_line(self, line):
        self._conversation.configure(state="normal")
        self._conversation.insert("end", line + "\n")
        self._conversation.configure(state="disabled")

    def set_status(self, text):
        self._status.configure(text=text)

    def set_input_enabled(self, enabled):
        state = "normal" if enabled else "disabled"
        self._input.configure(state=state)
        self._send.configure(state=state)


class RecordingView:
    """A view for tests: it only remembers what it was asked to show."""

    def __init__(self):
        self.calls = []

    def show_line(self, line):
        self.calls.append(line)

    def set_status(self, status):
        self.calls.append("[status] " + status)

    def set_input_enabled(self, enabled):
        pass                                             # not recorded


def repaint_delay(network_on_ui_thread, port):
    """How long a "repaint" waits when the network call runs on the UI thread itself."""
    fake_ui_thread = ThreadPoolExecutor(max_workers=1)
    blocking_call = lambda: safe_talk(port, 0.5, PoliteUser(1))   # waits 0.5 s for a
silent server
    if network_on_ui_thread:
        fake_ui_thread.submit(blocking_call)             # wrong: the UI thread is busy
waiting
    else:
        threading.Thread(target=blocking_call).start()   # right: a separate thread waits
    clicked = time.monotonic()
    fake_ui_thread.submit(lambda: None).result()         # the repaint the user is waiting
for
    waited = time.monotonic() - clicked
    fake_ui_thread.shutdown()
```

Listing 16.18 (Continued) *knock_knock_gui.py: the same design with Tkinter; the network thread posts view updates into a queue that the Tk thread drains.*

```
    return waited


class TestGui(unittest.TestCase):
    def test_one_joke_through_the_controller(self):
        server = KnockKnockServer(classics())
        view = RecordingView()
        Controller(view, lambda f: f(), PoliteUser(1)).connect(server.start(0)).join()
        server.stop()
        self.assertEqual(10, len(view.calls))
        self.assertEqual("[status] Conversation over", view.calls[9])

    def test_server_down_is_shown(self):
        view = RecordingView()
        Controller(view, lambda f: f(), PoliteUser(1)).connect(port_with_nobody()).join()
        self.assertEqual("[status] Cannot reach the server: is it running?", view.calls[1])

    def test_ui_thread_blocked_only_by_the_wrong_design(self):
        self.assertGreaterEqual(repaint_delay(True, bad_server([], False)), 0.4)
        self.assertLess(repaint_delay(False, bad_server([], False)), 0.4)


def run_window(port):
    import tkinter as tk
    root, answers = tk.Tk(), queue.Queue()
    view, jobs = TkView(root, answers), queue.Queue()

    def drain():                                        # Tk is not thread-safe: the Tk
thread
        while not jobs.empty():                         # runs every job the network thread
posted
            jobs.get()()
        root.after(50, drain)

    Controller(view, jobs.put, lambda line: answers.get()).connect(port)
    drain()
    root.mainloop()
```

Listing 16.18 (Continued) *knock_knock_gui.py: the same design with Tkinter; the network thread posts view updates into a queue that the Tk thread drains.*

```
if __name__ == "__main__":
    if len(sys.argv) > 1 and sys.argv[1] == "--selftest":
        server = KnockKnockServer(classics())
        view = RecordingView()
        Controller(view, lambda f: f(), PoliteUser(1)).connect(server.start(0)).join()
        server.stop()
        print("\n".join(view.calls))
        sys.stdout.flush()
        unittest.main(argv=sys.argv[:1], verbosity=2)
    else:
        run_window(int(sys.argv[1]) if len(sys.argv) > 1 else 4444)
```

```
[status] Connecting...
[status] Connected
Server: Knock! Knock!
You: Who's there?
Server: Lettuce
You: Lettuce who?
Server: Lettuce in, it's cold out here! Want another? (y/n)
You: n
Server: Bye.
[status] Conversation over
test_one_joke_through_the_controller
(__main__.TestGui.test_one_joke_through_the_controller) ... ok
test_server_down_is_shown (__main__.TestGui.test_server_down_is_shown) ... ok
test_ui_thread_blocked_only_by_the_wrong_design
(__main__.TestGui.test_ui_thread_blocked_only_by_the_wrong_design) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.503s

OK
```

Line by line. `connect` creates the `network` thread and returns at once, so a button handler that calls it finishes in microseconds. In `talk`, every interaction with the view is `ui.execute(() -> ...)`. The user function passed to `KnockKnockClient.converse` does three things: it posts "show the server's line and enable the input", then calls `replies.apply(line)`, which in the window *waits* on the queue until the user presses **Send**, and then posts "disable the input and show what the user said". That wait happens on the network thread, never on the event dispatch thread. The `catch` clauses turn failures into status messages, reusing the reasoning of Problem 16.6. In `SwingView`, the action listener of **Send** does the least possible work: it puts the text into the queue and clears the field. `input.addActionListener(e -> send.doClick())` makes the Enter key behave like the button. `main` builds the window inside `SwingUtilities.invokeLater`, because even creating Swing components must happen on the event dispatch thread.

The self-test records ten view calls: two statuses, the seven lines of a one-joke conversation, the `Bye.` shown at the end, and the closing status. A second controller, pointed at a port where nobody listens, shows "Cannot reach the server: is it running?" as its second call.

The measurement `repaintDelay` simulates the event dispatch thread with a single-thread executor. In the wrong design, the blocking network call is submitted to that same thread; the "repaint" submitted right after it has to wait until the call gives up after 500 ms. In the right design the call runs on its own thread and the repaint runs immediately. The printed comparisons are booleans ("at least 400 ms: true/false") rather than raw times, so the output is the same on every machine.

A tempting wrong solution. The natural first attempt puts the socket code straight into the **Send** button's action listener: read the server's line, append it, done. It even works when the server is fast. But while the listener waits for the network, the event dispatch thread can do nothing else: no repainting, no clicks, no closing the window. With a slow or silent server the window turns grey and the operating system offers to kill it. `repaintDelay(true, ...)` is that design in miniature, and it blocks the "repaint" for the full half second.

> **CHECK YOURSELF.** In `talk`, why does the code write `String last = t.get(t.size() - 1);` before `ui.execute(...)`, instead of calling `t.get(...)` inside the lambda?
>
> *Answer.* Both would work here, but computing `last` first keeps the lambda small and makes the data it carries to the other thread explicit and immutable (a `String`). A lambda may only

> use local variables that are effectively final; handing another thread only immutable values is the safest habit, and it is the same reasoning that made the joke store immutable.

16.6.3 Problem 16.8 — Performance criteria, product documentation, and the report

> **PROBLEM.** Describe the design from specification through design to implementation, and write the product documentation: a user-level guide, performance criteria, installation instructions and troubleshooting. Support the performance criteria with measurements. Outline the project report, including each member's job description and share of the work.

In our own words. Documentation is the part of a project that the team is most tempted to write last and in a hurry. The one part of it that must not be invented is the performance criteria: a promise such as "fifty users at once" is only worth something if it was measured. So we first write a small measuring program, and then the documents around its results.

Design. A *performance criterion* is a measurable promise with a number and a condition: "with 50 clients starting at the same moment, every conversation of one joke completes within 1000 ms on a laptop". Two quantities describe a server under load. The *latency* of one conversation is the time from connecting to hearing `Bye.`; we report the median, which half of the conversations beat, and the slowest one. The *throughput* is

$$X = \frac{n}{t} \tag{16.2}$$

conversations per second, where n conversations complete in t seconds of wall-clock time. Equation (16.2) and the latency are connected by a simple law of queues. If on average L conversations are in progress at once and each takes W seconds, then

$$L = X \cdot W \tag{16.3}$$

which is *Little's law* [3]. Equation (16.3) is a useful sanity check on any measurement: with a throughput of 900 conversations per second and a median latency of about 5 ms, about $900 \times 0.005 = 4.5$ conversations are in progress at a typical moment, which fits a machine with two processor cores that is also running the fifty client threads.

Measurements vary from run to run: the operating system schedules other work, the Java just-in-time compiler optimizes code while it runs, and creating threads costs time. Two habits make the numbers meaningful: a *warm-up* round that is not counted, and *repetition*, several rounds whose results are pooled.

The Java solution. **Listing 16.19** measures one, ten and fifty simultaneous clients, five rounds each, after a warm-up of twenty.

Listing 16.19 `PerformanceProbe.java`: latency and throughput of the knock-knock server with 1, 10 and 50 simultaneous clients.

```
// PerformanceProbe.java - Problem 16.8: measuring the numbers that the documentation
promises.
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class PerformanceProbe {

    static final int ROUNDS = 5;

    /** Runs n clients at the same instant; returns each client's time in milliseconds,
sorted. */
    static List<Double> latencies(int port, int n) throws Exception {
        ExecutorService pool = Executors.newFixedThreadPool(n);
        CountDownLatch startGate = new CountDownLatch(1);
        List<Future<Double>> futures = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            futures.add(pool.submit(() -> {
                startGate.await();
                long t0 = System.nanoTime();
                List<String> t = KnockKnockClient.talkTo("localhost", port,
                                                         new
KnockKnockClient.PoliteUser(1));
                if (!t.get(t.size() - 1).equals("S: Bye.")) {
                    throw new IllegalStateException("conversation failed");
                }
                return (System.nanoTime() - t0) / 1e6;
            }));
        }
        startGate.countDown();
        List<Double> result = new ArrayList<>();
        for (Future<Double> f : futures) {
            result.add(f.get());
        }
        pool.shutdown();
        Collections.sort(result);
        return result;
    }
```

Listing 16.19 (Continued) *PerformanceProbe.java: latency and throughput of the knock-knock server with 1, 10 and 50 simultaneous clients.*

```
    public static void main(String[] args) throws Exception {
        KnockKnockServer server = new KnockKnockServer(JokeStore.classics());
        int port = server.start(0);
        latencies(port, 20);                                // warm-up: let the JIT compile
the code
        System.out.println(" clients   median ms   slowest ms   conversations/s");
        double slowestAt50 = 0;
        for (int n : new int[] {1, 10, 50}) {
            List<Double> ms = new ArrayList<>();
            long t0 = System.nanoTime();
            for (int round = 0; round < ROUNDS; round++) {  // one measurement is not
enough
                ms.addAll(latencies(port, n));
            }
            double wallSeconds = (System.nanoTime() - t0) / 1e9;
            Collections.sort(ms);
            double median = ms.get(ms.size() / 2);
            double slowest = ms.get(ms.size() - 1);
            System.out.printf("%8d %11.1f %12.1f %17.0f%n", n, median, slowest,
                              n * ROUNDS / wallSeconds);
            slowestAt50 = slowest;
        }
        server.stop();
        System.out.println("criterion 'every one of 50 clients within 1000 ms' met: " +
(slowestAt50 < 1000));
    }
}
```

```
 clients   median ms   slowest ms   conversations/s
       1         1.7          2.1               488
      10         3.1         27.3               571
      50         4.7         59.9               905
criterion 'every one of 50 clients within 1000 ms' met: true
```

The Python solution. **Listing 16.20** is the same measurement. `time.perf_counter()` is Python's clock for measuring short intervals.

Listing 16.20 `performance_probe.py`: the same measurement in Python.

```
# performance_probe.py - Problem 16.8: measuring the numbers that the documentation
promises.
import threading
import time
from concurrent.futures import ThreadPoolExecutor
from joke_store import classics
from knock_knock_server import KnockKnockServer
from knock_knock_client import PoliteUser, talk_to

ROUNDS = 5


def latencies(port, n):
    """Runs n clients at the same instant; returns each client's time in milliseconds,
sorted."""
    start_gate = threading.Event()

    def one_client():
        start_gate.wait()
        t0 = time.perf_counter()
        t = talk_to("localhost", port, PoliteUser(1))
        if t[-1] != "S: Bye.":
            raise RuntimeError("conversation failed")
        return (time.perf_counter() - t0) * 1000

    with ThreadPoolExecutor(max_workers=n) as pool:
        futures = [pool.submit(one_client) for _ in range(n)]
        start_gate.set()
        return sorted(f.result() for f in futures)


if __name__ == "__main__":
    server = KnockKnockServer(classics())
    port = server.start(0)
    latencies(port, 20)                                  # warm-up
    print(" clients   median ms   slowest ms   conversations/s")
    for n in (1, 10, 50):
        ms, t0 = [], time.perf_counter()
        for _ in range(ROUNDS):                          # one measurement is not enough
            ms += latencies(port, n)
        wall_seconds = time.perf_counter() - t0
        ms.sort()
        median, slowest = ms[len(ms) // 2], ms[-1]
        print(f"{n:8d} {median:11.1f} {slowest:12.1f} {n * ROUNDS / wall_seconds:17.0f}")
    server.stop()
    print("criterion 'every one of 50 clients within 1000 ms' met:", slowest < 1000)
```

```
 clients   median ms   slowest ms   conversations/s
       1         4.2          8.9              149
      10         5.3         28.8              304
      50        16.0         42.1              701
criterion 'every one of 50 clients within 1000 ms' met: True
```

Your numbers will differ from these, and so will a second run on the same machine; what should not differ is the last line. Notice in the real output that the throughput *rises* from 1 to 50 clients in both languages: a single client pays the full cost of a new connection and a new server thread for every conversation and leaves the processor idle while it waits for the network, whereas fifty simultaneous clients keep both processor cores of the build machine busy. Such effects cannot be guessed; they have to be measured.

Line by line. `latencies` reuses the start gate of Problem 16.5, records `System.nanoTime()` before and after each conversation, and returns the sorted times in milliseconds (dividing nanoseconds by 10^6). The main loop pools five rounds for each client count, sorts them, takes the middle

element as the median and the last as the slowest, and computes Equation (16.2) with n equal to the number of conversations in all five rounds. `printf("%8d %11.1f ...")` prints fixed-width columns: `%8d` is an integer in 8 characters and `%11.1f` a number with one decimal in 11. The last line states the criterion and whether this run met it.

The documentation. With the measurements in hand, the product documentation is a set of short, concrete documents. **Table 16.7** is the outline the Knock-Knock Club uses; a real project replaces the contents but keeps the headings.

Table 16.7 The product documentation of the Knock-Knock Club, one row per document.

Document	Contents for the Knock-Knock Club
Specification	the goal, the user stories of Problem 16.1, **Table 16.5**
Design	**Figure 16.1**, **Figure 16.4**, the contracts `JokeStore`, the protocol and `View`
User guide	"Start the window, wait for *Connected*, answer the server's line in the box and press Send or Enter; answer `n` to leave."
Performance criteria	"50 simultaneous one-joke conversations, each within 1000 ms; measured with `PerformanceProbe`."
Installation	install JDK 17+ or Python 3.10+; copy the folder; run `javac *.java` once; start the server, then the clients
Troubleshooting	**Table 16.8**
Test plan	contract tests (16.1), protocol tests (16.2), cross tests with stubs (16.4), concurrency test (16.5), failure tests (16.6), headless GUI tests (16.7), performance probe (16.8)

A troubleshooting table lists symptoms that users actually see, in their words, with the cause and the cure. **Table 16.8** holds the entries that the failure scenarios of Problem 16.6 and the everyday use of sockets make likely.

Table 16.8 Troubleshooting the Knock-Knock Club: symptoms as the user sees them, causes, and remedies.

Symptom	Likely cause	What to do
"Cannot reach the server: is it running?"	server not started, wrong host or port, firewall	start `KnockKnockServer`; check host and port; allow the port in the firewall
server prints `Address already in use` (Java `BindException`, Python `OSError`)	another program, or an old copy of the server, holds the port	stop the old copy, or start the server on another port: `java KnockKnockServer 5555`
"The server did not answer within 500 ms."	server overloaded, stuck, or across a slow network	try again; check the server's console; raise the limit for slow networks
"The server closed the connection"	the server was stopped or crashed	restart the server; report its console output
window stays grey or does not respond	network code running on the user-interface thread	a programming error: see Problem 16.7

The report. A project report that someone else can read and check has a fixed skeleton. (1) *Introduction*: the goal and the non-goals. (2) *Specification*: the user stories and the contracts. (3) *Design*: the diagrams, the protocol table and the reasons for the main decisions (why one thread per client, why an immutable store). (4) *Implementation*: the structure of the code, with pseudocode or short excerpts for the interesting parts. (5) *Testing*: the test plan and the real test output. (6) *Evaluation*: the measured performance against the criteria, and what does not work yet. (7) *Conclusion*: what the team learned and would do differently. (8) *Contributions*: for each member, the parts owned, the work done, and an agreed percentage of the whole, for example "member 1, 33%: `JokeStore` and its two implementations, the contract test, the installation

guide". Writing the contributions as the project goes, as Step 10 suggests, makes them accurate and uncontroversial.

A tempting wrong solution. The tempting criterion is a single measurement written as a fact: "the server handles 50 users in 2 ms", taken from one lucky run. The output above shows why it is misleading: from run to run the slowest of the fifty conversations varied between about 15 and 70 ms on the same machine. A criterion needs a condition ("50 simultaneous clients"), a bound with a margin ("within 1000 ms"), and a method that anyone can repeat ("measured with `PerformanceProbe`, five rounds after a warm-up").

> **CHECK YOURSELF.** Suppose a measurement shows a throughput of 1000 conversations per second while, on average, 20 conversations are in progress. What average latency does Equation (16.3) predict?
>
> *Answer.* $W = L/X = 20/1000 = 0.02$ seconds, that is 20 ms per conversation.

16.7 New Worked Problems: Iterators and Immutable Values

The remaining six problems are new, each set in the running case study of the book, Profitina Laundry. The first two practice Chapter 13.

16.7.1 Problem 16.9 — A pickup schedule with its own iterator

> **PROBLEM.** Profitina Laundry's couriers collect orders at given hours. Write a class `PickupSchedule` that stores pickups (an order identifier and an hour) in an array of its own, always sorted by hour, with pickups at equal hours kept in the order they were added. Make it usable in a for-each loop, give it a *lazy* view of the pickups between two hours, and make its iterator *fail fast* if the schedule is changed during a loop.

In our own words. Java's for-each loop and Python's `for` loop work with any object that can produce an *iterator*: an object that hands out the elements one at a time and remembers where it is. We will write that object ourselves. "Lazy" means that the view does no work until someone iterates: it does not build a new list, it finds each matching pickup when the loop asks for the next one. "Fail fast" means that a loop over a schedule that someone changes in the middle stops at once with an exception, instead of quietly skipping or repeating elements.

Design. The schedule keeps an array `slots`, a count `size`, and a *modification count* `modCount` that grows by one at every change. The iterator is an *inner class*, `Cursor`, so each iterator object can see the fields of its schedule; it keeps its own position `next` and a copy of `modCount` taken when the loop started. Every call of `next()` compares the two counts. The lazy view `between(from, to)` returns an `Iterable` whose iterator wraps a normal iterator and skips non-matching pickups. Adding keeps the array sorted by the insertion step of insertion sort: shift later pickups one place right, then drop the new one in the gap.

The Java solution. Listing 16.21 contains the schedule, the cursor, the lazy view, and a wrong design in which the collection is its own iterator.

Listing 16.21 `PickupSchedule.java`: a sorted array with an inner-class iterator, a lazy filtered view and a fail-fast check.

```
// PickupSchedule.java - Problem 16.9: a collection with its own iterator, a lazy view, and
a fail-fast check.
import java.util.ArrayList;
import java.util.Arrays;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.List;
import java.util.NoSuchElementException;

public class PickupSchedule implements Iterable<PickupSchedule.Pickup> {

    /** One courier pickup: which order, at which hour of the day. */
    record Pickup(String orderId, int hour) {
    }

    private Pickup[] slots = new Pickup[2];
    private int size = 0;
    private int modCount = 0;                       // how many times the schedule has
changed

    /** Adds a pickup, keeping the schedule sorted by hour (equal hours keep their order).
*/
    public void add(Pickup p) {
        if (size == slots.length) {
            slots = Arrays.copyOf(slots, 2 * size);             // grow, as in Chapter 8
        }
        int i = size;
        while (i > 0 && slots[i - 1].hour() > p.hour()) {       // shift later pickups
right
            slots[i] = slots[i - 1];
            i--;
        }
        slots[i] = p;
        size++;
        modCount++;
    }

    /** Removes the pickup of one order; returns false if there was none. */
    public boolean removeOrder(String orderId) {
        for (int i = 0; i < size; i++) {
            if (slots[i].orderId().equals(orderId)) {
                System.arraycopy(slots, i + 1, slots, i, size - i - 1);
                slots[--size] = null;
                modCount++;
                return true;
            }
        }
        return false;
    }
```

Listing 16.21 (Continued) *PickupSchedule.java: a sorted array with an inner-class iterator, a lazy filtered view and a fail-fast check.*

```
    @Override
    public Iterator<Pickup> iterator() {
        return new Cursor();                        // a NEW position for every loop
    }

    /** The iterator: an inner class, so it can see slots, size and modCount. */
    private final class Cursor implements Iterator<Pickup> {
        private int next = 0;
        private final int expectedModCount = modCount;

        @Override
        public boolean hasNext() {
            return next < size;
        }

        @Override
        public Pickup next() {
            if (modCount != expectedModCount) {
                throw new ConcurrentModificationException("schedule changed during
iteration");
            }
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            return slots[next++];
        }
    }

    /** A lazy view: the pickups with from <= hour < to, found only while someone iterates.
*/
    public Iterable<Pickup> between(int from, int to) {
        return () -> new Iterator<>() {
            private final Iterator<Pickup> all = iterator();
            private Pickup ahead = advance();               // the next match, or null

            private Pickup advance() {
                while (all.hasNext()) {
                    Pickup p = all.next();
                    if (p.hour() >= from && p.hour() < to) {
                        return p;
                    }
                }
                return null;
            }
```

Listing 16.21 (Continued) *PickupSchedule.java: a sorted array with an inner-class iterator, a lazy filtered view and a fail-fast check.*

```
            @Override
            public boolean hasNext() {
                return ahead != null;
            }

            @Override
            public Pickup next() {
                if (ahead == null) {
                    throw new NoSuchElementException();
                }
                Pickup p = ahead;
                ahead = advance();
                return p;
            }
        };
    }

    /** Tempting but wrong: the collection IS its own iterator, so it has only one
position. */
    static final class SelfIteratingSchedule implements Iterable<String>, Iterator<String>
{
        private final List<String> ids;
        private int next = 0;

        SelfIteratingSchedule(List<String> ids) {
            this.ids = ids;
        }

        @Override
        public Iterator<String> iterator() {
            return this;                              // every loop shares the same "next"
        }

        @Override
        public boolean hasNext() {
            return next < ids.size();
        }

        @Override
        public String next() {
            return ids.get(next++);
        }
    }

    private static int passed = 0;
```

Listing 16.21 (Continued) *PickupSchedule.java: a sorted array with an inner-class iterator, a lazy filtered view and a fail-fast check.*

```
    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        PickupSchedule s = new PickupSchedule();
        s.add(new Pickup("A1", 14));
        s.add(new Pickup("A2", 9));
        s.add(new Pickup("A3", 17));
        s.add(new Pickup("A4", 11));
        s.add(new Pickup("A5", 9));
        List<String> all = new ArrayList<>();
        for (Pickup p : s) {                          // the for-each loop calls iterator()
            all.add(p.orderId() + "@" + p.hour());
        }
        System.out.println("whole day : " + all);
        List<String> morning = new ArrayList<>();
        for (Pickup p : s.between(8, 12)) {
            morning.add(p.orderId());
        }
        System.out.println("morning   : " + morning);

        int pairs = 0;
        for (Pickup a : s) {
            for (Pickup b : s) {
                pairs++;
            }
        }
        SelfIteratingSchedule wrong = new SelfIteratingSchedule(List.of("A1", "A2", "A3",
"A4", "A5"));
        int wrongPairs = 0;
        for (String a : wrong) {
            for (String b : wrong) {
                wrongPairs++;
            }
        }
        System.out.println("pairs     : " + pairs + " (self-iterating version: " +
wrongPairs + ")");
```

Listing 16.21 (Continued) *PickupSchedule.java: a sorted array with an inner-class iterator, a lazy filtered view and a fail-fast check.*

```
        String failure = "none";
        try {
            for (Pickup p : s) {
                if (p.hour() == 11) {
                    s.removeOrder(p.orderId());     // changing the schedule inside the
loop
                }
            }
        } catch (ConcurrentModificationException e) {
            failure = e.getMessage();
        }
        System.out.println("remove in loop: " + failure);

        check("sorted by hour, stable", "[A2@9, A5@9, A4@11, A1@14, A3@17]",
all.toString());
        check("lazy view", List.of("A2", "A5", "A4"), morning);
        check("nested loops", 25, pairs);
        check("self-iterating loses pairs", 4, wrongPairs);
        check("fail-fast", "schedule changed during iteration", failure);
        Iterator<Pickup> it = new PickupSchedule().iterator();
        check("empty schedule", false, it.hasNext());
        try {
            it.next();
            throw new AssertionError("no exception");
        } catch (NoSuchElementException e) {
            check("next past the end", "NoSuchElementException",
e.getClass().getSimpleName());
        }
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
whole day : [A2@9, A5@9, A4@11, A1@14, A3@17]
morning   : [A2, A5, A4]
pairs     : 25 (self-iterating version: 4)
remove in loop: schedule changed during iteration
PASS sorted by hour, stable
PASS lazy view
PASS nested loops
PASS self-iterating loses pairs
PASS fail-fast
PASS empty schedule
PASS next past the end
All 7 checks passed.
```

The Python solution. In Python, `__iter__` must return an iterator, and an iterator is any object with a `__next__` method that raises `StopIteration` at the end. **Listing 16.22** writes the cursor out by hand to show the protocol, and writes the lazy view as a *generator*, a function containing `yield`, for which Python writes the iterator class itself.

Listing 16.22 `pickup_schedule.py`: the same schedule with an explicit iterator class and a generator for the lazy view.

```
# pickup_schedule.py - Problem 16.9: a collection with its own iterator, a lazy view, and a
fail-fast check.
import sys
import unittest
from dataclasses import dataclass


@dataclass(frozen=True)
class Pickup:
    """One courier pickup: which order, at which hour of the day."""
    order_id: str
    hour: int


class PickupSchedule:
    def __init__(self):
        self._slots = []
        self._mod_count = 0                     # how many times the schedule has changed

    def add(self, pickup):
        """Adds a pickup, keeping the schedule sorted by hour (equal hours keep their
order)."""
        i = len(self._slots)
        while i > 0 and self._slots[i - 1].hour > pickup.hour:
            i -= 1
        self._slots.insert(i, pickup)
        self._mod_count += 1

    def remove_order(self, order_id):
        """Removes the pickup of one order; returns False if there was none."""
        for i, p in enumerate(self._slots):
            if p.order_id == order_id:
                del self._slots[i]
                self._mod_count += 1
                return True
        return False

    def __len__(self):
        return len(self._slots)

    def __iter__(self):
        return _Cursor(self)                    # a NEW position for every loop
```

Listing 16.22 (Continued) *pickup_schedule.py: the same schedule with an explicit iterator class and a generator for the lazy view.*

```
    def between(self, start, end):
        """A lazy view: a generator finds the pickups with start <= hour < end
        only while someone iterates."""
        for p in self:
            if start <= p.hour < end:
                yield p


class _Cursor:
    """The iterator, written out by hand: __iter__ and __next__."""

    def __init__(self, schedule):
        self._schedule = schedule
        self._next = 0
        self._expected = schedule._mod_count

    def __iter__(self):
        return self

    def __next__(self):
        if self._schedule._mod_count != self._expected:
            raise RuntimeError("schedule changed during iteration")
        if self._next >= len(self._schedule._slots):
            raise StopIteration                # Python's way of saying "no more"
        p = self._schedule._slots[self._next]
        self._next += 1
        return p


class SelfIteratingSchedule:
    """Tempting but wrong: the collection IS its own iterator, so it has only one
position."""

    def __init__(self, ids):
        self._ids = list(ids)
        self._next = 0

    def __iter__(self):
        return self                            # every loop shares the same _next

    def __next__(self):
        if self._next >= len(self._ids):
            raise StopIteration
        self._next += 1
        return self._ids[self._next - 1]
```

Listing 16.22 (Continued) *pickup_schedule.py: the same schedule with an explicit iterator class and a generator for the lazy view.*

```
def sample():
    s = PickupSchedule()
    for order_id, hour in (("A1", 14), ("A2", 9), ("A3", 17), ("A4", 11), ("A5", 9)):
        s.add(Pickup(order_id, hour))
    return s


def count_pairs(collection):
    return sum(1 for _a in collection for _b in collection)


class TestSchedule(unittest.TestCase):
    def test_sorted_by_hour_and_stable(self):
        self.assertEqual(["A2@9", "A5@9", "A4@11", "A1@14", "A3@17"],
                         [f"{p.order_id}@{p.hour}" for p in sample()])

    def test_lazy_view(self):
        self.assertEqual(["A2", "A5", "A4"], [p.order_id for p in sample().between(8, 12)])

    def test_nested_loops(self):
        self.assertEqual(25, count_pairs(sample()))
        self.assertEqual(4, count_pairs(SelfIteratingSchedule(["A1", "A2", "A3", "A4",
"A5"])))

    def test_fail_fast(self):
        s = sample()
        with self.assertRaises(RuntimeError):
            for p in s:
                if p.hour == 11:
                    s.remove_order(p.order_id)

    def test_next_past_the_end(self):
        with self.assertRaises(StopIteration):
            next(iter(PickupSchedule()))


if __name__ == "__main__":
    s = sample()
    print("whole day :", [f"{p.order_id}@{p.hour}" for p in s])
    print("morning   :", [p.order_id for p in s.between(8, 12)])
    print("pairs     :", count_pairs(s), "(self-iterating version:",
          str(count_pairs(SelfIteratingSchedule(["A1", "A2", "A3", "A4", "A5"]))) + ")")
    view = s.between(8, 12)                      # a generator object is used up after one
loop
    print("one generator, two loops:", len(list(view)), "then", len(list(view)))
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
whole day : ['A2@9', 'A5@9', 'A4@11', 'A1@14', 'A3@17']
morning   : ['A2', 'A5', 'A4']
pairs     : 25 (self-iterating version: 4)
one generator, two loops: 3 then 0
test_fail_fast (__main__.TestSchedule.test_fail_fast) ... ok
test_lazy_view (__main__.TestSchedule.test_lazy_view) ... ok
test_nested_loops (__main__.TestSchedule.test_nested_loops) ... ok
test_next_past_the_end (__main__.TestSchedule.test_next_past_the_end) ... ok
test_sorted_by_hour_and_stable (__main__.TestSchedule.test_sorted_by_hour_and_stable) ...
ok

----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

Line by line. In `add`, the `while` loop moves every pickup with a *later* hour one slot to the right; the comparison is `>` and not `>=`, which is what keeps equal hours in their arrival order: A2 and A5 are both at 9 and stay in that order. When the array is full, `Arrays.copyOf` doubles it, the growth rule of Problem 8.17. `iterator()` returns `new Cursor()` every time, so every loop gets its own position; that is what makes the nested loops count $5 \times 5 = 25$ pairs. The `Cursor` records `expectedModCount` when it is created; `next()` throws `ConcurrentModificationException` when the schedule's `modCount` has changed since, which happens in the output when the loop removes A4. The anonymous iterator of `between` looks one element *ahead*: `ahead` always holds the next match or `null`, so `hasNext()` is simply `ahead != null`, and `next()` returns `ahead` and moves on. The view never copies the schedule, and if the schedule changes between two loops, the next loop sees the change.

In Python, `_Cursor.__next__` raises `StopIteration`, which the `for` statement catches silently to end the loop. The generator `between` contains `for p in self: if ...: yield p`; calling it runs no code at all, it only creates a generator object, and each `next` runs the body up to the following `yield`. The last line printed before the tests shows the one important difference: a generator object *is* an iterator, so it can be used for exactly one loop. `len(list(view))` is 3 the first time and 0 the second. `s.between(8, 12)` creates a fresh generator each time it is called, which is why the schedule's own loops never run into this.

A tempting wrong solution. `SelfIteratingSchedule` implements `Iterable` and `Iterator` at once, and `iterator()` returns `this`. It is fewer lines, and a single loop works. But every loop over it shares one position. In the nested loops, the outer loop takes A1, the inner loop runs through A2 to A5 and exhausts the shared position, and when the inner loop ends the outer loop finds nothing left: 4 pairs instead of 25. For a collection of n elements, nested loops visit

$$P_{\text{own}} = n^2, \qquad P_{\text{shared}} = n - 1 \tag{16.4}$$

pairs with one iterator per loop and with a shared one; Equation (16.4) with $n = 5$ gives the 25 and the 4 of the output. A collection must hand out a *new* iterator for every loop.

> **CHECK YOURSELF.** In the fail-fast loop, the code removes A4, which is the third of five pickups. What would happen if the loop removed the pickup at 14 o'clock (A1, the fourth) instead?
>
> *Answer.* After A1 is removed, `size` is 4 and the cursor's `next` is also 4, so `hasNext()` returns `false` and the loop ends normally, without an exception and without visiting A3. Java's own `ArrayList` iterator behaves in exactly the same way for the second-to-last element. This is why the check is called "best effort": fail-fast iterators catch most mistakes, but correct code must never rely on the exception. To remove during a loop, use the iterator's own `remove` or `removeIf` (Exercise 16.2).

16.7.2 Problem 16.10 — Money as an immutable value object

> **PROBLEM.** Write a class `Money` for Profitina Laundry: an amount in whole units and a three-letter currency code. A `Money` object must never change after it is created; adding two amounts or pricing a weight must produce new objects. Two `Money` objects with the same amount and currency must be equal and must work as the same key in a hash map. Adding different currencies must be refused.

In our own words. Some objects have an identity: two customers named Sari are two people. Others are only a value: two banknotes of Rp 20,000 are interchangeable. A *value object* is a class of the second kind, and three things make it behave like a number: it is immutable, `equals`

compares the values and not the object identities, and `hashCode` agrees with `equals`. The last point is where most home-made value classes fail, silently.

> **DEFINITION.** The *hash contract*: if `a.equals(b)` is true, then `a.hashCode() == b.hashCode()` must be true as well. A hash map finds a key by first computing its hash code and looking only in that bucket; a key whose hash code disagrees with `equals` is looked for in the wrong bucket.

Design. The class is `final` so that no subclass can add a mutable field; its two fields are `private final`; the constructor is private and the static factory `of` validates the currency (three capital letters) and the amount (not negative), so that *every* `Money` in existence is valid. `plus` and `times` return new objects. `equals` and `hashCode` use exactly the same two fields. Amounts are whole units stored in a `long`, never a `double`, for the reason of Problem 8.18: binary floating point cannot represent most decimal fractions exactly.

The Java solution. **Listing 16.23** shows the class, the same value object as a *record*, and two broken variants.

Listing 16.23 `Money.java`: an immutable value object with validated construction, value equality, a consistent hash code, ordering, and two broken variants.

```java
// Money.java - Problem 16.10: an immutable value object with equals, hashCode and
compareTo.
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.Objects;
import java.util.Set;
import java.util.TreeMap;

public final class Money implements Comparable<Money> {    // final: no subclass can add a
setter

    private final String currency;                  // "IDR"
    private final long amount;                      // whole units: rupiah have no cents in
daily use

    private Money(String currency, long amount) {
        this.currency = currency;
        this.amount = amount;
    }

    /** The only way in: every Money that exists has passed these checks. */
    public static Money of(long amount, String currency) {
        if (currency == null || !currency.matches("[A-Z]{3}")) {
            throw new IllegalArgumentException("currency must be three capital letters: " +
currency);
        }
        if (amount < 0) {
            throw new IllegalArgumentException("amount must not be negative: " + amount);
        }
        return new Money(currency, amount);
    }

    public static Money rupiah(long amount) {
        return of(amount, "IDR");
    }

    public long amount() {
        return amount;
    }

    public String currency() {
        return currency;
    }
```

Listing 16.23 (Continued) *Money.java: an immutable value object with validated construction, value equality, a consistent hash code, ordering, and two broken variants.*

```
    /** Returns a NEW Money; this one never changes. */
    public Money plus(Money other) {
        requireSameCurrency(other);
        return new Money(currency, Math.addExact(amount, other.amount));   // overflow
throws
    }

    /** The price of kg kilograms at this price per kilogram, rounded to whole units. */
    public Money times(double kg) {
        return new Money(currency, Math.round(amount * kg));
    }

    private void requireSameCurrency(Money other) {
        if (!currency.equals(other.currency)) {
            throw new IllegalArgumentException(currency + " and " + other.currency + " do
not mix");
        }
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Money)) {
            return false;
        }
        Money m = (Money) o;
        return amount == m.amount && currency.equals(m.currency);
    }

    @Override
    public int hashCode() {
        return Objects.hash(currency, amount);      // built from exactly the fields equals
uses
    }

    @Override
    public int compareTo(Money other) {
        requireSameCurrency(other);
        return Long.compare(amount, other.amount);
    }

    @Override
    public String toString() {
        return String.format(Locale.ROOT, "%s %,d", currency, amount);
    }
```

Listing 16.23 (Continued) *Money.java: an immutable value object with validated construction, value equality, a consistent hash code, ordering, and two broken variants.*

```
    /** The same value object as a record: Java writes equals, hashCode and toString for
us. */
    record MoneyRecord(String currency, long amount) {
        MoneyRecord {
            if (amount < 0) {
                throw new IllegalArgumentException("amount must not be negative: " +
amount);
            }
        }
    }

    /** Tempting but wrong (1): equals without hashCode. */
    static final class NoHashMoney {
        final long amount;

        NoHashMoney(long amount) {
            this.amount = amount;
        }

        @Override
        public boolean equals(Object o) {
            return o instanceof NoHashMoney && ((NoHashMoney) o).amount == amount;
        }
    }

    /** Tempting but wrong (2): a key that can change after it went into a set. */
    static final class MutableMoney {
        long amount;

        MutableMoney(long amount) {
            this.amount = amount;
        }

        @Override
        public boolean equals(Object o) {
            return o instanceof MutableMoney && ((MutableMoney) o).amount == amount;
        }

        @Override
        public int hashCode() {
            return Long.hashCode(amount);
        }
    }

    private static int passed = 0;
```

Listing 16.23 (Continued) *Money.java: an immutable value object with validated construction, value equality, a consistent hash code, ordering, and two broken variants.*

```
    private static void check(String name, Object expected, Object actual) {
        if (!Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) {
        Money a = Money.rupiah(24_500);
        Money b = Money.rupiah(7_000).times(3.5);           // 3.5 kg of washing
        System.out.println("a = " + a + ", b = " + b);
        System.out.println("a == b: " + (a == b) + ", a.equals(b): " + a.equals(b)
                           + ", same hash: " + (a.hashCode() == b.hashCode()));

        Map<Money, Integer> ordersPerPrice = new HashMap<>();
        for (long price : new long[] {21_000, 24_500, 21_000, 70_000, 21_000}) {
            ordersPerPrice.merge(Money.rupiah(price), 1, Integer::sum);
        }
        System.out.println("orders per price: " + new TreeMap<>(ordersPerPrice));

        Set<NoHashMoney> noHash = new HashSet<>(List.of(new NoHashMoney(5000)));
        noHash.add(new NoHashMoney(5000));
        System.out.println("equals without hashCode, set size: " + noHash.size());

        MutableMoney key = new MutableMoney(5000);
        Set<MutableMoney> mutable = new HashSet<>();
        mutable.add(key);
        key.amount = 9000;                                  // changed while inside the set
        System.out.println("mutable key: contains " + mutable.contains(key) + ", size " +
mutable.size());

        check("value equality", a, b);
        check("hash agrees with equals", a.hashCode(), b.hashCode());
        check("plus makes a new object", "IDR 45,500",
a.plus(Money.rupiah(21_000)).toString());
        check("a is unchanged", "IDR 24,500", a.toString());
        check("three orders at 21,000", 3, ordersPerPrice.get(Money.rupiah(21_000)));
        check("record equality", new MoneyRecord("IDR", 5000), new MoneyRecord("IDR",
5000));
        try {
            a.plus(Money.of(10, "USD"));
            throw new AssertionError("no exception");
        } catch (IllegalArgumentException e) {
            check("currencies do not mix", "IDR and USD do not mix", e.getMessage());
        }
        check("the ghost key cannot be found", false, mutable.contains(key));
        System.out.println("All " + passed + " checks passed.");
    }
```

Listing 16.23 (Continued) *Money.java: an immutable value object with validated construction, value equality, a consistent hash code, ordering, and two broken variants.*

```
}
```

```
a = IDR 24,500, b = IDR 24,500
a == b: false, a.equals(b): true, same hash: true
orders per price: {IDR 21,000=3, IDR 24,500=1, IDR 70,000=1}
equals without hashCode, set size: 2
mutable key: contains false, size 1
PASS value equality
PASS hash agrees with equals
PASS plus makes a new object
PASS a is unchanged
PASS three orders at 21,000
PASS record equality
PASS currencies do not mix
PASS the ghost key cannot be found
All 8 checks passed.
```

The Python solution. **Listing 16.24** uses a frozen dataclass, which writes `__eq__` and `__hash__` from the fields and forbids assignment.

Listing 16.24 `money.py`: the same value object as a frozen dataclass, and the two broken variants.

```
# money.py - Problem 16.10: an immutable value object with equality, hashing and ordering.
import dataclasses
import re
import sys
import unittest
from collections import Counter
from dataclasses import dataclass


@dataclass(frozen=True)                         # frozen: assignment raises an error
class Money:
    amount: int                                 # whole units: rupiah have no cents in
daily use
    currency: str = "IDR"

    def __post_init__(self):                    # every Money that exists has passed these
checks
        if not re.fullmatch(r"[A-Z]{3}", self.currency or ""):
            raise ValueError(f"currency must be three capital letters: {self.currency}")
        if self.amount < 0:
            raise ValueError(f"amount must not be negative: {self.amount}")

    def __add__(self, other):
        """Returns a NEW Money; this one never changes."""
        self._require_same_currency(other)
        return Money(self.amount + other.amount, self.currency)

    def times(self, kg):
        """The price of kg kilograms at this price per kilogram, rounded to whole units."""
        return Money(round(self.amount * kg), self.currency)

    def __lt__(self, other):
        self._require_same_currency(other)
        return self.amount < other.amount

    def _require_same_currency(self, other):
        if self.currency != other.currency:
            raise ValueError(f"{self.currency} and {other.currency} do not mix")

    def __str__(self):
        return f"{self.currency} {self.amount:,}"


class NoHashMoney:
    """Tempting but wrong (1): __eq__ without __hash__."""
```

Listing 16.24 (Continued) *money.py: the same value object as a frozen dataclass, and the two broken variants.*

```
    def __init__(self, amount):
        self.amount = amount

    def __eq__(self, other):
        return isinstance(other, NoHashMoney) and other.amount == self.amount


@dataclass(unsafe_hash=True)                    # hashable AND changeable: asking for
trouble
class MutableMoney:
    """Tempting but wrong (2): a key that can change after it went into a set."""
    amount: int


class TestMoney(unittest.TestCase):
    def test_value_equality_and_hash(self):
        a, b = Money(24_500), Money(7_000).times(3.5)
        self.assertEqual(a, b)
        self.assertEqual(hash(a), hash(b))

    def test_plus_makes_a_new_object(self):
        a = Money(24_500)
        self.assertEqual("IDR 45,500", str(a + Money(21_000)))
        self.assertEqual("IDR 24,500", str(a))

    def test_cannot_change(self):
        with self.assertRaises(dataclasses.FrozenInstanceError):
            Money(1).amount = 2

    def test_currencies_do_not_mix(self):
        with self.assertRaises(ValueError):
            Money(24_500) + Money(10, "USD")

    def test_counting_by_price(self):
        counts = Counter(Money(p) for p in (21_000, 24_500, 21_000, 70_000, 21_000))
        self.assertEqual(3, counts[Money(21_000)])

    def test_ghost_key(self):
        key = MutableMoney(5000)
        keys = {key}
        key.amount = 9000
        self.assertNotIn(key, keys)
```

Listing 16.24 (Continued) *money.py: the same value object as a frozen dataclass, and the two broken variants.*

```
if __name__ == "__main__":
    a, b = Money(24_500), Money(7_000).times(3.5)       # 3.5 kg of washing
    print(f"a = {a}, b = {b}")
    print(f"a is b: {a is b}, a == b: {a == b}, same hash: {hash(a) == hash(b)}")
    counts = Counter(Money(p) for p in (21_000, 24_500, 21_000, 70_000, 21_000))
    print("orders per price:", {str(m): n for m, n in sorted(counts.items())})
    try:
        {NoHashMoney(5000)}
    except TypeError as e:
        print("__eq__ without __hash__:", e)
    key = MutableMoney(5000)
    keys = {key}
    key.amount = 9000                                   # changed while inside the set
    print(f"mutable key: contains {key in keys}, size {len(keys)}")
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
a = IDR 24,500, b = IDR 24,500
a is b: False, a == b: True, same hash: True
orders per price: {'IDR 21,000': 3, 'IDR 24,500': 1, 'IDR 70,000': 1}
__eq__ without __hash__: unhashable type: 'NoHashMoney'
mutable key: contains False, size 1
test_cannot_change (__main__.TestMoney.test_cannot_change) ... ok
test_counting_by_price (__main__.TestMoney.test_counting_by_price) ... ok
test_currencies_do_not_mix (__main__.TestMoney.test_currencies_do_not_mix) ... ok
test_ghost_key (__main__.TestMoney.test_ghost_key) ... ok
test_plus_makes_a_new_object (__main__.TestMoney.test_plus_makes_a_new_object) ... ok
test_value_equality_and_hash (__main__.TestMoney.test_value_equality_and_hash) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.000s

OK
```

Line by line. `Money.rupiah(7_000).times(3.5)` computes 7000 × 3.5 = 24,500 and builds a *new* object; `a == b` is `false` because they are two objects, while `a.equals(b)` is `true` because they hold the same value, and their hash codes agree because `Objects.hash(currency, amount)` is computed from the same fields that `equals` compares. `Math.addExact` throws `ArithmeticException` instead of silently wrapping around if a sum ever exceeds the range of `long`. `ordersPerPrice.merge(key, 1, Integer::sum)` counts orders per price: it puts 1 for a new key and adds 1 for an existing one, which works only because equal prices are equal keys. `String.format(Locale.ROOT, "%s %,d", ...)` prints the amount with thousands separators independently of the computer's language settings. The record `MoneyRecord` shows how much of this Java now writes for you: a *compact constructor* for validation, and nothing else. The printed lines also show that `compareTo` is used by the `TreeMap`, which sorts the prices from 21,000 to 70,000.

In Python, `__post_init__` runs after the generated `__init__` and is the place for validation. `round(self.amount * kg)` differs from Java's `Math.round` in one respect: Python rounds exact halves to the nearest *even* number (`round(2.5)` is 2), Java rounds them up. For money, write the rounding rule in the specification and test it. `Counter` is a dictionary that counts; it relies on `__hash__` exactly as `HashMap` relies on `hashCode`. Assigning to a field of a frozen dataclass raises `FrozenInstanceError`, which a test checks.

A tempting wrong solution. There are two, and the output shows both. `NoHashMoney` overrides `equals` but not `hashCode`; in Java, two equal objects then have different identity hash codes, land

in different buckets, and a `HashSet` happily holds two "equal" elements (`set size: 2`). `javac -Xlint` even warns about this class. Python refuses the mistake outright: a class that defines `__eq__` without `__hash__` gets `__hash__ = None` and cannot be put into a set at all (`unhashable type`). `MutableMoney` has both methods, but its amount can change; after it is changed inside a set, the set looks in the bucket of the *new* hash code, does not find it, and reports `contains false` while its size is still 1: a ghost element that can be neither found nor removed. Keys of hash maps and elements of hash sets must be immutable, which is the strongest practical argument for value objects.

> **CHECK YOURSELF.** Is `Money.rupiah(5000).equals(Money.of(5000, "USD"))` true? Is `Money.rupiah(0)` allowed? What does `Money.rupiah(-1)` do?
>
> *Answer.* False: the currencies differ, and `equals` compares both fields. `rupiah(0)` is allowed, since only negative amounts are refused. `rupiah(-1)` throws `IllegalArgumentException("amount must not be negative: -1")`, so no invalid `Money` ever exists.

16.8 New Worked Problems: Threads

16.8.1 Problem 16.11 — A shared counter, a measured race, and its cure

> **PROBLEM.** Four washing machines run as four threads and each adds 1 to a shared counter for every load it washes. Measure how many updates an unsynchronized counter loses, show a way to make the loss happen every time, and fix the counter in three different ways in Java and one in Python.

In our own words. Chapter 14 explained the *race condition*: `count++` looks like one step, but it is three, read the value, add one, write it back, and two threads can interleave those steps. **Figure 16.7** shows the smallest possible accident. The expected final count is

$$N = T \cdot k \tag{16.5}$$

for T threads that each add k times, and every interleaving like the one in the figure makes the real count c smaller, so the number of *lost updates* is

$$L_{\text{lost}} = N - c. \tag{16.6}$$

step	machine 1	count in memory	machine 2
1	read count: 41	41	
2		41	read count: 41
3	write 41 + 1 = 42	42	
4		42	write 41 + 1 = 42

Two loads were washed, but the count went up by one: one update is lost.

Figure 16.7 A lost update. Both machines read 41 before either writes, so both write 42: two loads were washed, and the counter rose by one.

Design. Five counters implement one interface, `Counter`, so one method `run` can test them all. `UnsafeCounter` is the bug; `SlowUnsafeCounter` is the same bug with the gap between read and write held open for 1 ms, which turns a rare accident into a certain one; `SynchronizedCounter`,

`LockedCounter` and `AtomicCounter` are three cures. With $T = 4$ and $k = 1{,}000{,}000$, Equation (16.5) gives $N = 4{,}000{,}000$.

The Java solution. Listing 16.25:

Listing 16.25 `WashCounter.java`: an unsafe counter, the same race made certain, and three thread-safe counters, all run by four threads released together.

```
// WashCounter.java - Problem 16.11: a shared counter, a measured race, and three ways to
fix it.
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicLong;

public class WashCounter {

    static final int THREADS = 4;
    static final int LOADS_PER_THREAD = 1_000_000;

    interface Counter {
        void addLoad();

        long total();
    }

    /**
     * Tempting but wrong: count++ is read, add, write, and two threads can interleave.
     * volatile makes every read and write go to main memory, but it does NOT make ++
atomic.
     */
    static final class UnsafeCounter implements Counter {
        private volatile long count = 0;

        public void addLoad() {
            count++;
        }

        public long total() {
            return count;
        }
    }

    /** The same bug with the window between read and write held open for 1 ms. */
    static final class SlowUnsafeCounter implements Counter {
        private long count = 0;

        public void addLoad() {
            long seen = count;                      // 1. read
            try {
                Thread.sleep(1);                    //    ... another machine runs now ...
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            count = seen + 1;                       // 2. write back: may undo its work
        }
```

Listing 16.25 (Continued) *WashCounter.java: an unsafe counter, the same race made certain, and three thread-safe counters, all run by four threads released together.*

```
        public long total() {
            return count;
        }
    }

    /** Fix 1: synchronized methods, so only one thread at a time is inside. */
    static final class SynchronizedCounter implements Counter {
        private long count = 0;

        public synchronized void addLoad() {
            count++;
        }

        public synchronized long total() {
            return count;
        }
    }

    /** Fix 2: a synchronized block on a private lock object that nobody else can see. */
    static final class LockedCounter implements Counter {
        private final Object lock = new Object();
        private long count = 0;

        public void addLoad() {
            synchronized (lock) {
                count++;
            }
        }

        public long total() {
            synchronized (lock) {
                return count;
            }
        }
    }

    /** Fix 3: an atomic variable, whose increment the hardware does in one indivisible
step. */
    static final class AtomicCounter implements Counter {
        private final AtomicLong count = new AtomicLong();

        public void addLoad() {
            count.incrementAndGet();
        }
```

Listing 16.25 (Continued) *WashCounter.java: an unsafe counter, the same race made certain, and three thread-safe counters, all run by four threads released together.*

```
        public long total() {
            return count.get();
        }
    }

    /** Starts THREADS washing machines that all add to one counter; waits for them with
join(). */
    static long run(Counter counter, int loadsPerThread) throws InterruptedException {
        Thread[] machines = new Thread[THREADS];
        CountDownLatch startGate = new CountDownLatch(1);
        for (int i = 0; i < THREADS; i++) {
            machines[i] = new Thread(() -> {
                try {
                    startGate.await();              // all machines start at the same
moment
                } catch (InterruptedException e) {
                    return;
                }
                for (int k = 0; k < loadsPerThread; k++) {
                    counter.addLoad();
                }
            });
            machines[i].start();
        }
        startGate.countDown();
        for (Thread m : machines) {
            m.join();                               // wait until this machine has finished
        }
        return counter.total();
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }
```

Listing 16.25 (Continued) *WashCounter.java: an unsafe counter, the same race made certain, and three thread-safe counters, all run by four threads released together.*

```
    public static void main(String[] args) throws InterruptedException {
        long expected = (long) THREADS * LOADS_PER_THREAD;
        System.out.println("expected total: " + expected);
        for (int trial = 1; trial <= 3; trial++) {
            long got = run(new UnsafeCounter(), LOADS_PER_THREAD);
            System.out.println("unsafe, trial " + trial + ": " + got + " (lost " +
(expected - got) + ")");
        }
        long slow = run(new SlowUnsafeCounter(), 50);
        System.out.println("slow unsafe: " + slow + " of " + THREADS * 50);
        Counter[] fixed = {new SynchronizedCounter(), new LockedCounter(), new
AtomicCounter()};
        for (Counter c : fixed) {
            long t0 = System.nanoTime();
            long got = run(c, LOADS_PER_THREAD);
            long ms = (System.nanoTime() - t0) / 1_000_000;
            System.out.println(c.getClass().getSimpleName() + ": " + got + " in " + ms + "
ms");
            check(c.getClass().getSimpleName() + " is exact", expected, got);
        }
        check("the widened window always loses updates", true, slow < THREADS * 50);
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
expected total: 4000000
unsafe, trial 1: 3992166 (lost 7834)
unsafe, trial 2: 3948364 (lost 51636)
unsafe, trial 3: 3939457 (lost 60543)
slow unsafe: 50 of 200
SynchronizedCounter: 4000000 in 63 ms
PASS SynchronizedCounter is exact
LockedCounter: 4000000 in 61 ms
PASS LockedCounter is exact
AtomicCounter: 4000000 in 43 ms
PASS AtomicCounter is exact
PASS the widened window always loses updates
All 4 checks passed.
```

The Python solution. **Listing 16.26** has the unsafe counter, the slow one, and the lock.

Listing 16.26 `wash_counter.py`: the same experiment in Python.

```
# wash_counter.py - Problem 16.11: a shared counter, a measured race, and a lock that fixes
it.
import sys
import threading
import time
import unittest

THREADS = 4
LOADS_PER_THREAD = 200_000


class UnsafeCounter:
    """Tempting but wrong: self.count += 1 is read, add, write, and threads can
interleave."""

    def __init__(self):
        self.count = 0

    def add_load(self):
        self.count += 1

    def total(self):
        return self.count


class SlowUnsafeCounter(UnsafeCounter):
    """The same bug with the window between read and write held open for 1 ms."""

    def add_load(self):
        seen = self.count                       # 1. read
        time.sleep(0.001)                       #    ... another machine runs now ...
        self.count = seen + 1                   # 2. write back: may undo its work


class LockedCounter:
    """The fix: a lock, so only one thread at a time is between read and write."""

    def __init__(self):
        self._lock = threading.Lock()
        self._count = 0

    def add_load(self):
        with self._lock:                        # acquired here, released at the end of the
block
            self._count += 1
```

Listing 16.26 (Continued) *wash_counter.py: the same experiment in Python.*

```
    def total(self):
        with self._lock:
            return self._count


def run(counter, loads_per_thread):
    """Starts THREADS washing machines that all add to one counter; waits for them with
join()."""
    start_gate = threading.Event()

    def machine():
        start_gate.wait()                       # all machines start at the same moment
        for _ in range(loads_per_thread):
            counter.add_load()

    machines = [threading.Thread(target=machine) for _ in range(THREADS)]
    for m in machines:
        m.start()
    start_gate.set()
    for m in machines:
        m.join()                                # wait until this machine has finished
    return counter.total()


class TestCounters(unittest.TestCase):
    def test_locked_counter_is_exact(self):
        self.assertEqual(THREADS * 1000, run(LockedCounter(), 1000))

    def test_widened_window_always_loses_updates(self):
        self.assertLess(run(SlowUnsafeCounter(), 50), THREADS * 50)


if __name__ == "__main__":
    sys.setswitchinterval(1e-6)                 # switch threads very often: races show up
sooner
    expected = THREADS * LOADS_PER_THREAD
    print("expected total:", expected)
    for trial in (1, 2, 3):
        got = run(UnsafeCounter(), LOADS_PER_THREAD)
        print(f"unsafe, trial {trial}: {got} (lost {expected - got})")
    print("slow unsafe:", run(SlowUnsafeCounter(), 50), "of", THREADS * 50)
    t0 = time.perf_counter()
    got = run(LockedCounter(), LOADS_PER_THREAD)
    print(f"LockedCounter: {got} in {(time.perf_counter() - t0) * 1000:.0f} ms")
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
expected total: 800000
unsafe, trial 1: 800000 (lost 0)
unsafe, trial 2: 800000 (lost 0)
unsafe, trial 3: 800000 (lost 0)
slow unsafe: 50 of 200
LockedCounter: 800000 in 189 ms
test_locked_counter_is_exact (__main__.TestCounters.test_locked_counter_is_exact) ... ok
test_widened_window_always_loses_updates
(__main__.TestCounters.test_widened_window_always_loses_updates) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.080s

OK
```

Line by line. `run` creates four threads, lets all of them wait on a start gate, opens it, and calls `join()` on each, which waits until that thread has finished; only then is `total()` read. In the Java

output, the unsafe counter lost between a few thousand and sixty thousand of four million updates in three trials: by Equation (16.6), the first trial lost $4{,}000{,}000 - 3{,}992{,}166 = 7{,}834$. On another run or another machine you may see more, fewer, or, occasionally, none: when the operating system happens to run the threads one after another, nothing interleaves. That is exactly what makes races so dangerous. The field is declared `volatile`, which forces every read and write to go to main memory and prevents the compiler from keeping the count in a register; many people believe that `volatile` fixes the race, and the output shows that it does not, because the three steps are still three steps.

`SlowUnsafeCounter` reads the count, sleeps for 1 ms, and writes the old value plus one. During the sleep the other three threads read the same value, so in each round all four write the same result: 50 of 200 remain, every time. Widening a race window on purpose is a real testing technique: it turns "fails once a month" into "fails in every test run", so that the fix can be verified.

The cures. `synchronized` on a method means that a thread must own the object's *monitor lock* to enter, so at most one thread at a time runs `addLoad` or `total` on that object. `LockedCounter` locks a private object instead, which is the same mechanism with one advantage: no code outside the class can take the same lock and interfere. `AtomicCounter` uses `AtomicLong.incrementAndGet()`, which the processor performs as one indivisible operation (a *compare-and-swap* loop), and it is typically the fastest of the three. All three reach exactly 4,000,000; the times printed after them vary between runs and should not be read too precisely.

The Python output holds a surprise, and it is real: the unsafe counter lost *nothing* in three trials, even though `sys.setswitchinterval(1e-6)` asks the interpreter to switch threads as often as it can. The reason is an implementation detail of CPython: the interpreter switches threads only at certain points of the bytecode, and in current versions the read and write of `self.count += 1` happen not to be separated by such a point. That is luck, not correctness: the language does not promise it, older and newer interpreters behave differently, and the *free-threaded* builds of Python 3.13 and later, which run without the global interpreter lock, interleave freely. The slow counter proves that the code is wrong regardless: 50 of 200, every time. `with self._lock:` acquires the lock at the start of the block and releases it at the end, even if an exception is raised, and makes the counter exact.

A tempting wrong solution. The tempting conclusion from the Python output is "`+=` is atomic in Python, so no lock is needed". A test that passes a thousand times does not prove that there is no race; it may only prove that this machine, this interpreter and this load never produced the bad interleaving. The correct reasoning is structural: if two threads read and write the same variable and at least one of them writes, the accesses must be protected, whether or not a test has ever failed.

> **CHECK YOURSELF.** In `SlowUnsafeCounter`, with 4 threads and 50 loads each, the result was 50. Explain why it is *exactly* 50, and what it would be with 2 threads.
>
> *Answer.* All four threads start together, read the same value, sleep, and write that value plus one, so each round adds 1 instead of 4; after 50 rounds the count is 50. (The rounds stay aligned because each sleep lasts the same time.) With 2 threads the result would be about 50 as well, now out of an expected 100.

16.8.2 Problem 16.12 — A producer–consumer wash line

> **PROBLEM.** The front desk of Profitina Laundry takes orders and puts them on a shelf that holds at most two orders; two washing machines take orders from the shelf and wash them.

> The desk must wait when the shelf is full and a machine must wait when it is empty. Write the shelf by hand with `wait` and `notifyAll` (Python: a `Condition`), then with the library's blocking queue. Stop the machines cleanly when the day's orders are done, and check that every order is washed exactly once.

In our own words. This is the *producer–consumer* problem of Chapter 14 [4]: one or more threads produce items, others consume them, and a *bounded buffer* between them smooths out differences in speed while keeping memory limited. The difficulty is waiting correctly: a thread that finds the buffer full or empty must sleep without holding the lock, and must be woken when the situation changes.

Design. Figure 16.8 shows the line. To stop the consumers, the producer puts one *poison pill*, a special value `STOP`, on the shelf for each machine after the last real order; a machine that takes a pill finishes. The hand-made buffer is a *monitor*: all its methods are `synchronized` on the buffer object, a full or empty buffer makes the caller `wait()`, which releases the lock while sleeping, and every change calls `notifyAll()`.

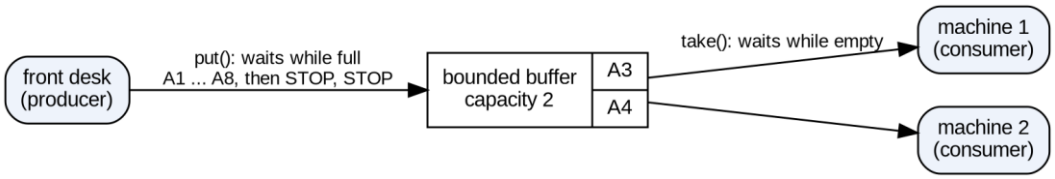


Figure 16.8 The wash line: one producer, a bounded buffer of capacity 2, and two consumers. Two `STOP` pills, one per machine, end the day.

The Java solution. Listing 16.27:

Listing 16.27 `WashLine.java`: a hand-made bounded buffer, the same line with `ArrayBlockingQueue`, poison pills, and a buffer that uses `if` instead of `while`.

```
// WashLine.java - Problem 16.12: producer-consumer with a bounded buffer, by hand and with
BlockingQueue.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Deque;
import java.util.HashSet;
import java.util.List;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class WashLine {

    static final String STOP = "STOP";              // the "poison pill": no more orders
    static final int MACHINES = 2;

    interface Buffer {
        void put(String order) throws InterruptedException;

        String take() throws InterruptedException;
    }

    /** A bounded buffer written by hand: a monitor with wait() and notifyAll(). */
    static final class OrderBuffer implements Buffer {
        private final Deque<String> items = new ArrayDeque<>();
        private final int capacity;
        private int mostEverWaiting = 0;

        OrderBuffer(int capacity) {
            this.capacity = capacity;
        }

        @Override
        public synchronized void put(String order) throws InterruptedException {
            while (items.size() == capacity) {
                wait();                             // full: the front desk waits for room
            }
            items.addLast(order);
            mostEverWaiting = Math.max(mostEverWaiting, items.size());
            notifyAll();                            // wake any machine waiting for work
        }
```

Listing 16.27 (Continued) *WashLine.java: a hand-made bounded buffer, the same line with ArrayBlockingQueue, poison pills, and a buffer that uses if instead of while.*

```
        @Override
        public synchronized String take() throws InterruptedException {
            while (items.isEmpty()) {
                wait();                              // empty: the machine waits for work
            }
            String order = items.removeFirst();
            notifyAll();                             // wake the desk if it waits for room
            return order;
        }

        synchronized int mostEverWaiting() {
            return mostEverWaiting;
        }
    }

    /** The library version: ArrayBlockingQueue already is a bounded buffer. */
    static Buffer libraryBuffer(int capacity) {
        BlockingQueue<String> queue = new ArrayBlockingQueue<>(capacity);
        return new Buffer() {
            @Override
            public void put(String order) throws InterruptedException {
                queue.put(order);                    // waits while the queue is full
            }

            @Override
            public String take() throws InterruptedException {
                return queue.take();                 // waits while the queue is empty
            }
        };
    }

    /** One front desk (producer) and MACHINES washing machines (consumers). */
    static List<String> runLine(Buffer buffer, int orders) throws InterruptedException {
        List<String> washed = Collections.synchronizedList(new ArrayList<>());
        Thread desk = new Thread(() -> {
            try {
                for (int i = 1; i <= orders; i++) {
                    buffer.put("A" + i);
                }
                for (int m = 0; m < MACHINES; m++) {
                    buffer.put(STOP);                // one pill for every machine
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
```

Listing 16.27 (Continued) *WashLine.java: a hand-made bounded buffer, the same line with ArrayBlockingQueue, poison pills, and a buffer that uses if instead of while.*

```
        List<Thread> threads = new ArrayList<>(List.of(desk));
        for (int m = 0; m < MACHINES; m++) {
            threads.add(new Thread(() -> {
                try {
                    while (true) {
                        String order = buffer.take();
                        if (order.equals(STOP)) {
                            return;                 // swallowed the pill: stop
                        }
                        Thread.sleep(1);            // washing takes a little time
                        washed.add(order);
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }));
        }
        threads.forEach(Thread::start);
        for (Thread t : threads) {
            t.join();
        }
        return washed;
    }

    /** Tempting but wrong: "if" instead of "while" around wait(). */
    static final class IfBuffer {
        private final Deque<String> items = new ArrayDeque<>();
        private int waiting = 0;

        synchronized void put(String order) {
            items.addLast(order);
            notifyAll();
        }

        synchronized String take() throws InterruptedException {
            if (items.isEmpty()) {                  // checked ONCE, before sleeping
                waiting++;
                wait();
                waiting--;
            }
            return items.removeFirst();             // the list may be empty again by now
        }
```

Listing 16.27 (Continued) *WashLine.java: a hand-made bounded buffer, the same line with ArrayBlockingQueue, poison pills, and a buffer that uses if instead of while.*

```
        synchronized int waiting() {
            return waiting;
        }
    }

    /** Two machines wait on an empty IfBuffer; one order arrives; what does each get? */
    static List<String> ifBufferRace() throws InterruptedException {
        IfBuffer buffer = new IfBuffer();
        List<String> results = Collections.synchronizedList(new ArrayList<>());
        List<Thread> machines = new ArrayList<>();
        for (int m = 0; m < 2; m++) {
            Thread t = new Thread(() -> {
                try {
                    results.add(buffer.take());
                } catch (RuntimeException e) {
                    results.add(e.getClass().getSimpleName());
                } catch (InterruptedException e) {
                    results.add("interrupted");
                }
            });
            machines.add(t);
            t.start();
        }
        while (buffer.waiting() < 2) {              // until both machines are inside
wait()
            Thread.sleep(5);
        }
        buffer.put("A1");                           // one order, and notifyAll() wakes
both
        for (Thread t : machines) {
            t.join(2000);
        }
        List<String> sorted = new ArrayList<>(results);
        Collections.sort(sorted);
        return sorted;
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }
```

Listing 16.27 (Continued) *WashLine.java: a hand-made bounded buffer, the same line with ArrayBlockingQueue, poison pills, and a buffer that uses if instead of while.*

```
    public static void main(String[] args) throws InterruptedException {
        OrderBuffer handMade = new OrderBuffer(2);
        List<String> washed = runLine(handMade, 8);
        System.out.println("hand-made buffer: " + washed.size() + " orders washed, "
                           + new HashSet<>(washed).size() + " different, most ever waiting
"
                           + handMade.mostEverWaiting());
        List<String> washed2 = runLine(libraryBuffer(2), 8);
        System.out.println("library buffer  : " + washed2.size() + " orders washed, "
                           + new HashSet<>(washed2).size() + " different");
        List<String> race = ifBufferRace();
        System.out.println("if instead of while: " + race);

        check("every order washed once", 8, new HashSet<>(washed).size());
        check("no order washed twice", 8, washed.size());
        check("capacity respected", true, handMade.mostEverWaiting() <= 2);
        check("library buffer too", 8, new HashSet<>(washed2).size());
        check("the if-buffer hands out an order that is gone",
              List.of("A1", "NoSuchElementException"), race);
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
hand-made buffer: 8 orders washed, 8 different, most ever waiting 2
library buffer  : 8 orders washed, 8 different
if instead of while: [A1, NoSuchElementException]
PASS every order washed once
PASS no order washed twice
PASS capacity respected
PASS library buffer too
PASS the if-buffer hands out an order that is gone
All 5 checks passed.
```

The Python solution. **Listing 16.28** uses `threading.Condition`, which is a lock and a waiting room in one object, and wraps `queue.Queue(maxsize=2)` as the library buffer.

Listing 16.28 `wash_line.py`: the same wash line in Python.

```
# wash_line.py - Problem 16.12: producer-consumer with a bounded buffer, by hand and with
queue.Queue.
import queue
import sys
import threading
import time
import unittest
from collections import deque

STOP = "STOP"                                   # the "poison pill": no more orders
MACHINES = 2


class OrderBuffer:
    """A bounded buffer written by hand: a Condition with wait() and notify_all()."""

    def __init__(self, capacity):
        self._items = deque()
        self._capacity = capacity
        self._changed = threading.Condition()   # a lock plus a waiting room
        self.most_ever_waiting = 0

    def put(self, order):
        with self._changed:
            while len(self._items) == self._capacity:
                self._changed.wait()            # full: the front desk waits for room
            self._items.append(order)
            self.most_ever_waiting = max(self.most_ever_waiting, len(self._items))
            self._changed.notify_all()          # wake any machine waiting for work

    def take(self):
        with self._changed:
            while not self._items:
                self._changed.wait()            # empty: the machine waits for work
            order = self._items.popleft()
            self._changed.notify_all()          # wake the desk if it waits for room
            return order


class LibraryBuffer:
    """The library version: queue.Queue already is a bounded buffer."""

    def __init__(self, capacity):
        self._queue = queue.Queue(maxsize=capacity)
```

Listing 16.28 (Continued) *wash_line.py: the same wash line in Python.*

```
    def put(self, order):
        self._queue.put(order)                  # waits while the queue is full

    def take(self):
        return self._queue.get()                # waits while the queue is empty


def run_line(buffer, orders):
    """One front desk (producer) and MACHINES washing machines (consumers)."""
    washed, lock = [], threading.Lock()

    def desk():
        for i in range(1, orders + 1):
            buffer.put(f"A{i}")
        for _ in range(MACHINES):
            buffer.put(STOP)                    # one pill for every machine

    def machine():
        while True:
            order = buffer.take()
            if order == STOP:
                return                          # swallowed the pill: stop
            time.sleep(0.001)                   # washing takes a little time
            with lock:
                washed.append(order)

    threads = [threading.Thread(target=desk)] + [threading.Thread(target=machine)
                                                 for _ in range(MACHINES)]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    return washed


class IfBuffer:
    """Tempting but wrong: "if" instead of "while" around wait()."""

    def __init__(self):
        self._items = deque()
        self._changed = threading.Condition()
        self.waiting = 0
```

Listing 16.28 (Continued) *wash_line.py: the same wash line in Python.*

```
    def put(self, order):
        with self._changed:
            self._items.append(order)
            self._changed.notify_all()

    def take(self):
        with self._changed:
            if not self._items:                 # checked ONCE, before sleeping
                self.waiting += 1
                self._changed.wait()
                self.waiting -= 1
            return self._items.popleft()        # the deque may be empty again by now


def if_buffer_race():
    """Two machines wait on an empty IfBuffer; one order arrives; what does each get?"""
    buffer, results = IfBuffer(), []

    def machine():
        try:
            results.append(buffer.take())
        except IndexError as e:
            results.append(type(e).__name__)

    machines = [threading.Thread(target=machine) for _ in range(2)]
    for m in machines:
        m.start()
    while buffer.waiting < 2:                   # until both machines are inside wait()
        time.sleep(0.005)
    buffer.put("A1")                            # one order, and notify_all() wakes both
    for m in machines:
        m.join(2)
    return sorted(results)


class TestWashLine(unittest.TestCase):
    def test_hand_made_buffer(self):
        buffer = OrderBuffer(2)
        washed = run_line(buffer, 8)
        self.assertEqual(8, len(washed))
        self.assertEqual(8, len(set(washed)))
        self.assertLessEqual(buffer.most_ever_waiting, 2)

    def test_library_buffer(self):
        self.assertEqual(8, len(set(run_line(LibraryBuffer(2), 8))))
```

Listing 16.28 (Continued) *wash_line.py: the same wash line in Python.*

```
    def test_if_buffer_hands_out_an_order_that_is_gone(self):
        self.assertEqual(["A1", "IndexError"], if_buffer_race())


if __name__ == "__main__":
    buffer = OrderBuffer(2)
    washed = run_line(buffer, 8)
    print(f"hand-made buffer: {len(washed)} orders washed, {len(set(washed))} different, "
          f"most ever waiting {buffer.most_ever_waiting}")
    washed2 = run_line(LibraryBuffer(2), 8)
    print(f"library buffer  : {len(washed2)} orders washed, {len(set(washed2))} different")
    print("if instead of while:", if_buffer_race())
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
hand-made buffer: 8 orders washed, 8 different, most ever waiting 2
library buffer  : 8 orders washed, 8 different
if instead of while: ['A1', 'IndexError']
test_hand_made_buffer (__main__.TestWashLine.test_hand_made_buffer) ... ok
test_if_buffer_hands_out_an_order_that_is_gone
(__main__.TestWashLine.test_if_buffer_hands_out_an_order_that_is_gone) ... ok
test_library_buffer (__main__.TestWashLine.test_library_buffer) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.014s

OK
```

Line by line. In `put`, `while (items.size() == capacity) wait();` is the heart of the monitor. `wait()` does three things atomically: it releases the buffer's lock, puts the thread to sleep, and, when the thread is woken, re-acquires the lock before returning. The `while` re-checks the condition after every wake-up, for two reasons: another thread may have changed the buffer between the notification and this thread getting the lock back, and Java allows *spurious wake-ups*, returns from `wait()` with no notification at all. `notifyAll()` wakes every waiting thread; `notify()` would wake only one, which may be a thread of the wrong kind (the desk, when a machine was needed), and the line could stall. `mostEverWaiting` records the largest number of orders ever on the shelf, and the check confirms that the capacity was never exceeded. The machines sleep for 1 ms per order to simulate washing, which lets the desk fill the shelf. The desk puts eight orders and then two pills; each machine loops on `take()` until it takes a pill. `new HashSet<>(washed).size()` is 8 and `washed.size()` is 8: no order was lost and none was washed twice.

`ArrayBlockingQueue` does all of this in the library: `put` waits while the queue is full and `take` waits while it is empty. In real programs, use it (or Python's `queue.Queue`); write the monitor yourself only to understand it, as here.

A tempting wrong solution. `IfBuffer` writes `if (items.isEmpty()) wait();`, checking the condition once. The scenario in `ifBufferRace` is set up to be deterministic: both machines call `take()` on the empty buffer and wait; the main thread polls `waiting()` until both are inside `wait()`, then puts one order, and `notifyAll()` wakes both. The first machine to get the lock takes A1. The second one returns from `wait()`, does *not* check again, and calls `removeFirst()` on an empty deque: `NoSuchElementException` in Java, `IndexError` in Python. The sorted result `[A1, NoSuchElementException]` is the same on every run. Always wait in a `while` loop.

> **CHECK YOURSELF.** The desk puts one `STOP` pill for each of the two machines. What happens if it puts only one?
>
> *Answer.* One machine takes the pill and stops; the other finds the shelf empty and waits forever, so `t.join()` in `runLine` never returns and the program hangs. The number of pills must equal the number of consumers. (An alternative is a shared "closed" flag that `take` checks in its `while` loop, with `notifyAll` when it is set.)

16.9 New Worked Problem: A Tiny Protocol of Our Own

16.9.1 Problem 16.13 — LAUNDRY/1, a request–reply protocol

> **PROBLEM.** Design and implement a small text protocol, `LAUNDRY/1`, with which a courier's phone could ask Profitina Laundry for prices. After connecting, the server greets with `LAUNDRY/1 READY`. The client may send `SERVICES`, `PRICE <service> <kg>` and `QUIT`; every

> request gets exactly one reply line starting with `OK` or `ERR`. Write the server, a client class, and tests including ten clients at once.

In our own words. The knock-knock protocol is a conversation led by the server. Most real protocols are the opposite: the client asks, the server answers, and each request stands alone. Such a *request–reply* protocol is much easier to design, test and extend, because the server needs no conversation state: the whole protocol can be a *pure function* from one request line to one reply line.

Design. Table 16.9 is the specification. The status word at the start of every reply lets a client distinguish success from failure without understanding the rest; the message after `ERR` is for people.

Table 16.9 The LAUNDRY/1 protocol. Command words are case-insensitive; every request line gets exactly one reply line.

Request	Reply on success	Replies on failure
(on connecting)	`LAUNDRY/1 READY`	none
`SERVICES`	`OK EXPRESS=14000 IRON=5000 WASH=7000`	none
`PRICE <service> <kg>`	`OK <price in rupiah>`	`ERR usage: ...`, `ERR unknown service ...`, `ERR weight is not a number: ...`, `ERR weight must be in (0, 50] kg`
`QUIT`	`OK BYE`, then the server closes	none
anything else	none	`ERR unknown command ...`

The Java solution. Listing 16.29 contains the pure function `reply`, the server, the client and a client with a classic networking mistake.

Listing 16.29 `LaundryProtocol.java`: the protocol as a pure function, a multi-client server, a client with a reply timeout, and a client that forgets to flush.

```
// LaundryProtocol.java - Problem 16.13: a tiny request-reply protocol, its server, its
client.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class LaundryProtocol {

    static final String GREETING = "LAUNDRY/1 READY";
    static final Map<String, Long> PRICE_PER_KG = Map.of("WASH", 7000L, "IRON", 5000L,
"EXPRESS", 14000L);

    /** The whole protocol as one pure function: one request line in, one reply line out.
*/
    static String reply(String request) {
        String[] words = request.trim().split("\\s+");
        String command = words[0].toUpperCase(Locale.ROOT);
        switch (command) {
            case "SERVICES":
                return "OK EXPRESS=14000 IRON=5000 WASH=7000";
            case "QUIT":
                return "OK BYE";
            case "PRICE":
                if (words.length != 3) {
                    return "ERR usage: PRICE <service> <kg>";
                }
                Long rate = PRICE_PER_KG.get(words[1].toUpperCase(Locale.ROOT));
                if (rate == null) {
                    return "ERR unknown service " + words[1];
                }
                double kg;
                try {
                    kg = Double.parseDouble(words[2]);
                } catch (NumberFormatException e) {
                    return "ERR weight is not a number: " + words[2];
                }
```

Listing 16.29 (Continued) *LaundryProtocol.java: the protocol as a pure function, a multi-client server, a client with a reply timeout, and a client that forgets to flush.*

```
                if (kg <= 0 || kg > 50) {
                    return "ERR weight must be in (0, 50] kg";
                }
                return "OK " + Math.round(rate * kg);
            default:
                return "ERR unknown command " + command;
        }
    }

    /** The server: greets, then answers every line with exactly one line. */
    static final class Server {
        private final ExecutorService workers = Executors.newCachedThreadPool();
        private ServerSocket listener;

        int start() throws IOException {
            listener = new ServerSocket(0);
            Thread acceptor = new Thread(() -> {
                try {
                    while (true) {
                        Socket c = listener.accept();
                        workers.execute(() -> serve(c));
                    }
                } catch (IOException e) {
                    // listener closed
                }
            });
            acceptor.setDaemon(true);
            acceptor.start();
            return listener.getLocalPort();
        }

        private void serve(Socket c) {
            try (c;
                 BufferedReader in = new BufferedReader(new InputStreamReader(
                         c.getInputStream(), StandardCharsets.UTF_8));
                 PrintWriter out = new PrintWriter(new OutputStreamWriter(
                         c.getOutputStream(), StandardCharsets.UTF_8), true)) {
                out.println(GREETING);
                String line;
                while ((line = in.readLine()) != null) {
                    String answer = line.isBlank() ? "ERR empty request" : reply(line);
                    out.println(answer);
                    if (answer.equals("OK BYE")) {
                        return;
                    }
                }
```

Listing 16.29 (Continued) *LaundryProtocol.java: the protocol as a pure function, a multi-client server, a client with a reply timeout, and a client that forgets to flush.*

```
            } catch (IOException e) {
                // the client vanished; nothing to clean up beyond the sockets
            }
        }

        void stop() throws IOException {
            listener.close();
            workers.shutdown();
        }
    }

    /** The client: one method per request, with a time limit on every reply. */
    static class Client implements AutoCloseable {
        protected final Socket socket;
        protected final BufferedReader in;
        protected PrintWriter out;

        Client(int port) throws IOException {
            socket = new Socket("localhost", port);
            socket.setSoTimeout(500);
            in = new BufferedReader(new InputStreamReader(socket.getInputStream(),
StandardCharsets.UTF_8));
            out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream(),
StandardCharsets.UTF_8), true);
            String hello = in.readLine();
            if (!GREETING.equals(hello)) {
                throw new IOException("not a LAUNDRY/1 server: " + hello);
            }
        }

        String request(String line) throws IOException {
            out.println(line);
            try {
                return in.readLine();
            } catch (SocketTimeoutException e) {
                return "(no reply within 500 ms)";
            }
        }

        @Override
        public void close() throws IOException {
            socket.close();
        }
    }
```

Listing 16.29 (Continued) *LaundryProtocol.java: the protocol as a pure function, a multi-client server, a client with a reply timeout, and a client that forgets to flush.*

```
    /** Tempting but wrong: a PrintWriter without autoflush keeps the request in its
buffer. */
    static final class NoFlushClient extends Client {
        NoFlushClient(int port) throws IOException {
            super(port);
            out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream(),
StandardCharsets.UTF_8));
        }
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) throws Exception {
        Server server = new Server();
        int port = server.start();
        List<String> script = List.of("SERVICES", "PRICE wash 3.5", "price EXPRESS 2",
                                      "PRICE DRYCLEAN 1", "PRICE WASH abc", "PRICE IRON
80", "HELLO", "QUIT");
        List<String> replies = new ArrayList<>();
        try (Client c = new Client(port)) {
            for (String request : script) {
                String answer = c.request(request);
                replies.add(answer);
                System.out.printf("%-18s -> %s%n", request, answer);
            }
        }
        ExecutorService crowd = Executors.newFixedThreadPool(10);
        List<Future<String>> prices = new ArrayList<>();
        for (int i = 1; i <= 10; i++) {
            String request = "PRICE WASH " + i;
            prices.add(crowd.submit(() -> {
                try (Client c = new Client(port)) {
                    return c.request(request);
                }
            }));
        }
        long sum = 0;
        for (Future<String> f : prices) {
            sum += Long.parseLong(f.get().substring(3));
        }
        crowd.shutdown();
```

Listing 16.29 (Continued) *LaundryProtocol.java: the protocol as a pure function, a multi-client server, a client with a reply timeout, and a client that forgets to flush.*

```
        System.out.println("ten clients at once, total of their prices: " + sum);
        try (Client wrong = new NoFlushClient(port)) {
            String answer = wrong.request("PRICE WASH 1");
            System.out.println("client without flush -> " + answer);
            check("missing flush means no reply", "(no reply within 500 ms)", answer);
        }
        server.stop();

        check("price of 3.5 kg washing", "OK 24500", replies.get(1));
        check("command words ignore case", "OK 28000", replies.get(2));
        check("unknown service", "ERR unknown service DRYCLEAN", replies.get(3));
        check("bad weight", "ERR weight is not a number: abc", replies.get(4));
        check("unknown command", "ERR unknown command HELLO", replies.get(6));
        check("sum of 1..10 kg at 7000", 385_000L, sum);
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
SERVICES           -> OK EXPRESS=14000 IRON=5000 WASH=7000
PRICE wash 3.5     -> OK 24500
price EXPRESS 2    -> OK 28000
PRICE DRYCLEAN 1   -> ERR unknown service DRYCLEAN
PRICE WASH abc     -> ERR weight is not a number: abc
PRICE IRON 80      -> ERR weight must be in (0, 50] kg
HELLO              -> ERR unknown command HELLO
QUIT               -> OK BYE
ten clients at once, total of their prices: 385000
client without flush -> (no reply within 500 ms)
PASS missing flush means no reply
PASS price of 3.5 kg washing
PASS command words ignore case
PASS unknown service
PASS bad weight
PASS unknown command
PASS sum of 1..10 kg at 7000
All 7 checks passed.
```

The Python solution. Listing 16.30:

Listing 16.30 `laundry_protocol.py`: the same protocol, server and clients in Python.

```
# laundry_protocol.py - Problem 16.13: a tiny request-reply protocol, its server, its
client.
import socket
import sys
import threading
import unittest
from concurrent.futures import ThreadPoolExecutor

GREETING = "LAUNDRY/1 READY"
PRICE_PER_KG = {"WASH": 7000, "IRON": 5000, "EXPRESS": 14000}


def reply(request):
    """The whole protocol as one pure function: one request line in, one reply line out."""
    words = request.split()
    command = words[0].upper()
    if command == "SERVICES":
        return "OK EXPRESS=14000 IRON=5000 WASH=7000"
    if command == "QUIT":
        return "OK BYE"
    if command == "PRICE":
        if len(words) != 3:
            return "ERR usage: PRICE <service> <kg>"
        rate = PRICE_PER_KG.get(words[1].upper())
        if rate is None:
            return f"ERR unknown service {words[1]}"
        try:
            kg = float(words[2])
        except ValueError:
            return f"ERR weight is not a number: {words[2]}"
        if not 0 < kg <= 50:
            return "ERR weight must be in (0, 50] kg"
        return f"OK {round(rate * kg)}"
    return f"ERR unknown command {command}"


class Server:
    """The server: greets, then answers every line with exactly one line."""

    def start(self):
        self._listener = socket.create_server(("localhost", 0))
        threading.Thread(target=self._accept_loop, daemon=True).start()
        return self._listener.getsockname()[1]
```

Listing 16.30 (Continued) *laundry_protocol.py: the same protocol, server and clients in Python.*

```
    def _accept_loop(self):
        try:
            while True:
                c, _ = self._listener.accept()
                threading.Thread(target=self._serve, args=(c,), daemon=True).start()
        except OSError:
            pass                                        # listener closed

    def _serve(self, c):
        try:
            with c, c.makefile("r", encoding="utf-8") as reader, \
                    c.makefile("w", encoding="utf-8") as writer:
                writer.write(GREETING + "\n")
                writer.flush()
                for line in reader:                     # ends when the client hangs up
                    answer = reply(line) if line.strip() else "ERR empty request"
                    writer.write(answer + "\n")
                    writer.flush()
                    if answer == "OK BYE":
                        return
        except OSError:
            pass                                        # the client vanished

    def stop(self):
        try:
            self._listener.shutdown(socket.SHUT_RDWR)
        except OSError:
            pass
        self._listener.close()


class Client:
    """The client: one method per request, with a time limit on every reply."""

    def __init__(self, port):
        self._socket = socket.create_connection(("localhost", port), timeout=0.5)
        self._reader = self._socket.makefile("r", encoding="utf-8")
        self._writer = self._socket.makefile("w", encoding="utf-8")
        hello = self._reader.readline().rstrip("\n")
        if hello != GREETING:
            raise OSError(f"not a LAUNDRY/1 server: {hello}")

    def _send(self, line):
        self._writer.write(line + "\n")
        self._writer.flush()                            # push the bytes onto the network
NOW
```

Listing 16.30 (Continued) *laundry_protocol.py: the same protocol, server and clients in Python.*

```
    def request(self, line):
        self._send(line)
        try:
            return self._reader.readline().rstrip("\n")
        except TimeoutError:
            return "(no reply within 500 ms)"

    def close(self):
        self._socket.close()

    def __enter__(self):
        return self

    def __exit__(self, *exc):
        self.close()


class NoFlushClient(Client):
    """Tempting but wrong: writes into the file buffer and never flushes it."""

    def _send(self, line):
        self._writer.write(line + "\n")


SCRIPT = ["SERVICES", "PRICE wash 3.5", "price EXPRESS 2", "PRICE DRYCLEAN 1",
          "PRICE WASH abc", "PRICE IRON 80", "HELLO", "QUIT"]


def ten_clients_at_once(port):
    def one(kg):
        with Client(port) as c:
            return int(c.request(f"PRICE WASH {kg}")[3:])
    with ThreadPoolExecutor(max_workers=10) as pool:
        return sum(pool.map(one, range(1, 11)))


class TestLaundryProtocol(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.server = Server()
        cls.port = cls.server.start()

    @classmethod
    def tearDownClass(cls):
        cls.server.stop()
```

Listing 16.30 (Continued) *laundry_protocol.py: the same protocol, server and clients in Python.*

```
    def test_script(self):
        with Client(self.port) as c:
            replies = [c.request(r) for r in SCRIPT]
        self.assertEqual("OK 24500", replies[1])
        self.assertEqual("OK 28000", replies[2])
        self.assertEqual("ERR unknown service DRYCLEAN", replies[3])
        self.assertEqual("ERR weight is not a number: abc", replies[4])
        self.assertEqual("ERR unknown command HELLO", replies[6])

    def test_ten_clients_at_once(self):
        self.assertEqual(385_000, ten_clients_at_once(self.port))

    def test_missing_flush_means_no_reply(self):
        with NoFlushClient(self.port) as wrong:
            self.assertEqual("(no reply within 500 ms)", wrong.request("PRICE WASH 1"))


if __name__ == "__main__":
    server = Server()
    port = server.start()
    with Client(port) as c:
        for request in SCRIPT:
            print(f"{request:<18} -> {c.request(request)}")
    print("ten clients at once, total of their prices:", ten_clients_at_once(port))
    with NoFlushClient(port) as wrong:
        print("client without flush ->", wrong.request("PRICE WASH 1"))
    server.stop()
    sys.stdout.flush()
    unittest.main(verbosity=2)
```

```
SERVICES           -> OK EXPRESS=14000 IRON=5000 WASH=7000
PRICE wash 3.5     -> OK 24500
price EXPRESS 2    -> OK 28000
PRICE DRYCLEAN 1   -> ERR unknown service DRYCLEAN
PRICE WASH abc     -> ERR weight is not a number: abc
PRICE IRON 80      -> ERR weight must be in (0, 50] kg
HELLO              -> ERR unknown command HELLO
QUIT               -> OK BYE
ten clients at once, total of their prices: 385000
client without flush -> (no reply within 500 ms)
test_missing_flush_means_no_reply
(__main__.TestLaundryProtocol.test_missing_flush_means_no_reply) ... ok
test_script (__main__.TestLaundryProtocol.test_script) ... ok
test_ten_clients_at_once (__main__.TestLaundryProtocol.test_ten_clients_at_once) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.529s

OK
```

Line by line. `reply` splits the request on runs of white space (`"\\s+"` in Java; `split()` without an argument in Python) and switches on the upper-cased first word, so `price EXPRESS 2` works. Each `ERR` branch corresponds to one row of **Table 16.9**. The price is `Math.round(rate * kg)`: $7000 \times 3.5 = 24{,}500$ and $14{,}000 \times 2 = 28{,}000$, as the output shows. Because `reply` is pure, most of the protocol could be tested with no sockets at all; the network test is there to check the framing and the concurrency. The server greets, then answers each line with exactly one line; a blank line gets `ERR empty request` rather than an exception from the parsing code. The `Client` reads and checks the greeting in its constructor, so a client connected to the wrong kind of server fails immediately with a clear message. `request` sends one line and reads one line with a 500 ms limit. Ten clients at once ask for the price of 1 to 10 kg of washing; the sum of the replies is $7000 \times (1 + 2 + \cdots + 10) = 7000 \times 55 = 385{,}000$, which the check confirms.

A tempting wrong solution. `NoFlushClient` differs from `Client` in one argument: its `PrintWriter` is created without autoflush. `println` then writes the request into the writer's internal buffer, not onto the network. The server never receives it, so it never replies; the client waits in `readLine()` and, thanks to the timeout, reports `(no reply within 500 ms)`. Without the timeout it would hang forever, and nothing in the code would look wrong. The Python `NoFlushClient` makes the same mistake by leaving out `self._writer.flush()`. Buffers make writing efficient, but a request–reply protocol must flush after every request.

> **CHECK YOURSELF.** Add a request `QUOTE <service> <kg>` that returns the same price as `PRICE` but with 10% added for delivery, rounded to whole rupiah. What does `QUOTE WASH 3.5` return, and where does the change go?
>
> *Answer.* `OK 26950` (24,500 × 1.1 = 26,950). The change goes only into `reply`: a new `case "QUOTE"` that computes the `PRICE` reply and, if it starts with `OK`, multiplies the number by 1.1 and rounds. Neither the server nor the client changes, which is the benefit of keeping the protocol a pure function.

16.10 The Capstone: Profitina Laundry from End to End

16.10.1 Problem 16.14 — A small but complete laundry system

> **PROBLEM.** Build a small Profitina Laundry system that combines the main ideas of the whole book. Customers place orders for a service (wash, iron or express) and a weight; each order gets an identifier and a price and moves through the statuses received, washing, ready and picked up. The shop reports its orders and its revenue, saves its orders to a file and loads them again after a restart. Several counters work at the same time through a socket API. Use classes, an interface, inheritance, collections, a checked exception, files, and the Model–View–Controller pattern, in Java and in Python, with tests.

In our own words. This problem is the book in miniature. The best way to read it is as a list of decisions, each of which some earlier chapter taught, and to recognize each one as it appears.

Design. **Figure 16.9** is the class diagram. The system has four layers, from the network inward: the `SocketApi` accepts connections and passes each command line to the `Controller`; the controller calls the `Shop`, the model, and turns results and refusals into reply lines; the shop holds the orders and tells its observers, the views, about every change; the orders, the services and the repository are the smallest building blocks. **Table 16.10** lists where each idea of the book appears.

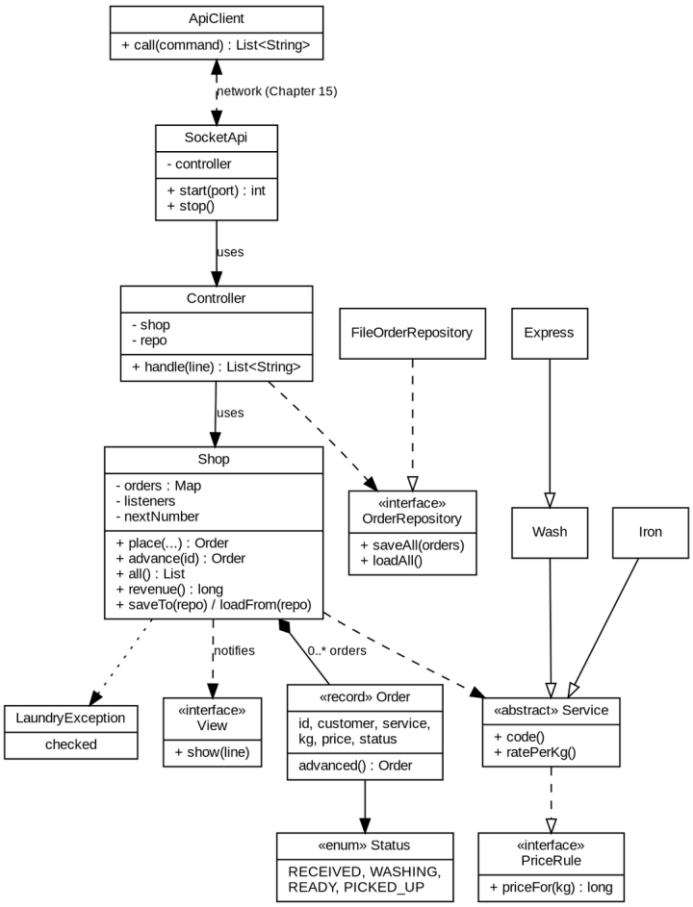


Figure 16.9 UML class diagram of the capstone. Hollow triangles mean “extends” (solid line) or “implements” (dashed line); the diamond means that the shop holds the orders.

Table 16.10 Where the ideas of the whole book appear in the capstone.

Idea of the book	Chapter	Where it appears in the capstone
classes, objects, strings	3	every class; `describe` formats an order
exceptions, files	5	checked `LaundryException`; `FileOrderRepository`
interfaces, documentation, tests	6	`PriceRule`, `OrderRepository`, `View`; self-tests
inheritance, polymorphism, `super`	7	`Service`, `Wash`, `Express`; `ratePerKg()` overridden
collections, generics	9	`LinkedHashMap<String, Order>`, `List<Consumer<Order>>`
events, observers	10	listeners called on every change
Model–View–Controller	11	`Shop`, `View`, `Controller`
immutability, enums	13	`Order` record, `advanced()`, `Status.next()`, snapshot lists
thread-safety	14	`synchronized` shop methods, `CopyOnWriteArrayList` of listeners
sockets, protocol framing	15	`SocketApi`, `ApiClient`, the lone-dot terminator

The socket API needs one new piece of protocol design. The command `LIST` returns one line per order, so replies now have a variable number of lines, and a client needs a rule to know where a reply ends. We use the rule of the mail protocol SMTP [5]: every reply ends with a line containing only a dot. No reply line can be a lone dot, because every order line starts with an identifier and every status line with `OK` or `ERR`.

The Java solution. The model is **Listing 16.31**, `LaundryCore.java`, a holder class for the model types. It has a private constructor, like `ListOps` in Chapter 8, because nobody should create a `LaundryCore` object.

Listing 16.31 `LaundryCore.java`: the model of the capstone: a checked exception, the `PriceRule` interface, the service hierarchy, the status enum, the immutable `Order` record, the file repository, and the thread-safe `Shop`.

```
// LaundryCore.java - Problem 16.14 (part 1): the model of Profitina Laundry.
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Collection;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.function.Consumer;

public final class LaundryCore {

    private LaundryCore() {
    }

    /** A checked exception: every caller must decide what a refused request means to it.
*/
    public static final class LaundryException extends Exception {
        private static final long serialVersionUID = 1L;

        public LaundryException(String message) {
            super(message);
        }
    }

    /** Anything that can price a weight (Chapter 6: an interface). */
    public interface PriceRule {
        long priceFor(double kg);
    }

    /** The services (Chapter 7: an abstract class and a small hierarchy). */
    public abstract static class Service implements PriceRule {
        public abstract String code();

        public abstract long ratePerKg();

        @Override
        public long priceFor(double kg) {
            return Math.round(ratePerKg() * kg);
        }
    }
```

Listing 16.31 (Continued) *LaundryCore.java: the model of the capstone: a checked exception, the PriceRule interface, the service hierarchy, the status enum, the immutable Order record, the file repository, and the thread-safe Shop.*

```
    public static class Wash extends Service {
        public String code() {
            return "WASH";
        }

        public long ratePerKg() {
            return 7000;
        }
    }

    public static final class Iron extends Service {
        public String code() {
            return "IRON";
        }

        public long ratePerKg() {
            return 5000;
        }
    }

    public static final class Express extends Wash {        // an express wash IS a wash
        @Override
        public String code() {
            return "EXPRESS";
        }

        @Override
        public long ratePerKg() {
            return 2 * super.ratePerKg();                   // twice the washing rate
        }
    }

    static final Map<String, Service> SERVICES = Map.of("WASH", new Wash(), "IRON", new
Iron(),
                                                        "EXPRESS", new Express());

    public static Service service(String code) throws LaundryException {
        Service s = SERVICES.get(code.toUpperCase(Locale.ROOT));
        if (s == null) {
            throw new LaundryException("unknown service " + code);
        }
        return s;
    }
```

Listing 16.31 (Continued) *LaundryCore.java: the model of the capstone: a checked exception, the PriceRule interface, the service hierarchy, the status enum, the immutable Order record, the file repository, and the thread-safe Shop.*

```
    /** The life of an order, in order. */
    public enum Status {
        RECEIVED, WASHING, READY, PICKED_UP;

        Status next() throws LaundryException {
            if (this == PICKED_UP) {
                throw new LaundryException("already picked up");
            }
            return values()[ordinal() + 1];
        }
    }

    /** One order (Chapter 13: immutable; a change makes a new Order). */
    public record Order(String id, String customer, String service, double kg, long price,
                        Status status) {
        Order advanced() throws LaundryException {
            return new Order(id, customer, service, kg, price, status.next());
        }

        String toLine() {
            return String.join(";", id, customer, service, Double.toString(kg),
                               Long.toString(price), status.name());
        }

        static Order fromLine(String line) {
            String[] p = line.split(";");
            return new Order(p[0], p[1], p[2], Double.parseDouble(p[3]),
Long.parseLong(p[4]),
                             Status.valueOf(p[5]));
        }
    }

    /** Where orders are kept between runs (Chapter 5: files, behind an interface). */
    public interface OrderRepository {
        void saveAll(Collection<Order> orders) throws IOException;

        List<Order> loadAll() throws IOException;
    }

    public static final class FileOrderRepository implements OrderRepository {
        private final Path file;

        public FileOrderRepository(Path file) {
            this.file = file;
        }
```

Listing 16.31 (Continued) *LaundryCore.java: the model of the capstone: a checked exception, the PriceRule interface, the service hierarchy, the status enum, the immutable Order record, the file repository, and the thread-safe Shop.*

```
        @Override
        public void saveAll(Collection<Order> orders) throws IOException {
            List<String> lines = new ArrayList<>();
            for (Order o : orders) {
                lines.add(o.toLine());
            }
            Files.write(file, lines);
        }

        @Override
        public List<Order> loadAll() throws IOException {
            List<Order> result = new ArrayList<>();
            if (Files.exists(file)) {
                for (String line : Files.readAllLines(file)) {
                    result.add(Order.fromLine(line));
                }
            }
            return result;
        }
    }

    /** The model (Chapter 11). Every public method is synchronized: many clients share one
shop. */
    public static final class Shop {
        private final Map<String, Order> orders = new LinkedHashMap<>();   // keeps arrival
order
        private final List<Consumer<Order>> listeners = new CopyOnWriteArrayList<>();
        private int nextNumber = 1;

        /** Observers (the V of MVC) are told about every change. */
        public void addListener(Consumer<Order> listener) {
            listeners.add(listener);
        }

        public synchronized Order place(String customer, String serviceCode, String kgText)
                throws LaundryException {
            if (customer.isBlank() || customer.contains(";")) {
                throw new LaundryException("customer name is missing or contains ';'");
            }
            Service s = service(serviceCode);
            double kg;
            try {
                kg = Double.parseDouble(kgText);
            } catch (NumberFormatException e) {
                throw new LaundryException("weight is not a number: " + kgText);
            }
            if (kg <= 0 || kg > 50) {
                throw new LaundryException("weight must be in (0, 50] kg");
```

Listing 16.31 (Continued) *LaundryCore.java: the model of the capstone: a checked exception, the PriceRule interface, the service hierarchy, the status enum, the immutable Order record, the file repository, and the thread-safe Shop.*

```
            }
            Order o = new Order("L" + nextNumber++, customer, s.code(), kg, s.priceFor(kg),
                                Status.RECEIVED);
            orders.put(o.id(), o);
            listeners.forEach(l -> l.accept(o));
            return o;
        }

        public synchronized Order advance(String id) throws LaundryException {
            Order old = orders.get(id);
            if (old == null) {
                throw new LaundryException("no order " + id);
            }
            Order changed = old.advanced();
            orders.put(id, changed);                        // replace, never modify
            listeners.forEach(l -> l.accept(changed));
            return changed;
        }

        public synchronized List<Order> all() {
            return List.copyOf(orders.values());            // a snapshot the caller cannot
change
        }

        public synchronized long revenue() {
            long sum = 0;
            for (Order o : orders.values()) {
                sum += o.price();
            }
            return sum;
        }

        public synchronized void saveTo(OrderRepository repo) throws IOException {
            repo.saveAll(orders.values());
        }

        public synchronized void loadFrom(OrderRepository repo) throws IOException {
            orders.clear();
            for (Order o : repo.loadAll()) {
                orders.put(o.id(), o);
                nextNumber = Math.max(nextNumber, Integer.parseInt(o.id().substring(1)) +
1);
            }
        }
    }
}
```

Listing 16.32, `ProfitinaLaundry.java`, adds the controller, the views and the socket API. Its `main` method has three modes: `java ProfitinaLaundry 5000 orders.csv` runs the server on port 5000 with a console view; `java ProfitinaLaundry client 5000` is a keyboard client; with no arguments it runs the self-test.

Listing 16.32 `ProfitinaLaundry.java`: the controller, the views, the socket API with dot-terminated replies, a client, and the self-test.

```
// ProfitinaLaundry.java - Problem 16.14 (part 2): controller, views and a socket API
around LaundryCore.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.Locale;
import java.util.Set;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class ProfitinaLaundry {

    static final String GREETING = "PROFITINA-LAUNDRY/1 READY";
    static final String END = ".";                  // a line with only a dot ends every
reply

    /** The V of MVC: anything that can show a line of text. */
    interface View {
        void show(String line);
    }

    /** The C of MVC: turns one command line into calls on the model and lines of reply. */
    static final class Controller {
        private final LaundryCore.Shop shop;
        private final LaundryCore.OrderRepository repo;

        Controller(LaundryCore.Shop shop, LaundryCore.OrderRepository repo) {
            this.shop = shop;
            this.repo = repo;
        }
```

Listing 16.32 (Continued) *ProfitinaLaundry.java: the controller, the views, the socket API with dot-terminated replies, a client, and the self-test.*

```
        List<String> handle(String commandLine) {
            String[] w = commandLine.trim().split("\\s+");
            try {
                switch (w[0].toUpperCase(Locale.ROOT)) {
                    case "PLACE":
                        need(w, 4, "PLACE <customer> <service> <kg>");
                        LaundryCore.Order o = shop.place(w[1], w[2], w[3]);
                        return List.of("OK " + o.id() + " " + o.price());
                    case "ADVANCE":
                        need(w, 2, "ADVANCE <id>");
                        return List.of("OK " + w[1] + " " + shop.advance(w[1]).status());
                    case "LIST":
                        List<String> lines = new ArrayList<>();
                        for (LaundryCore.Order x : shop.all()) {
                            lines.add(describe(x));
                        }
                        lines.add("OK " + lines.size() + " orders");
                        return lines;
                    case "REVENUE":
                        return List.of("OK " + shop.revenue());
                    case "SAVE":
                        shop.saveTo(repo);
                        return List.of("OK saved " + shop.all().size() + " orders");
                    default:
                        return List.of("ERR unknown command " + w[0]);
                }
            } catch (LaundryCore.LaundryException e) {
                return List.of("ERR " + e.getMessage());   // a refused request, not a
crash
            } catch (IOException e) {
                return List.of("ERR storage failed: " + e.getClass().getSimpleName());
            }
        }

        private static void need(String[] w, int n, String usage) throws
LaundryCore.LaundryException {
            if (w.length != n) {
                throw new LaundryCore.LaundryException("usage: " + usage);
            }
        }
    }

    static String describe(LaundryCore.Order o) {
        return String.format(Locale.ROOT, "%s %s %s %.1f kg IDR %,d %s", o.id(),
o.customer(),
                             o.service(), o.kg(), o.price(), o.status());
    }
```

Listing 16.32 (Continued) *ProfitinaLaundry.java: the controller, the views, the socket API with dot-terminated replies, a client, and the self-test.*

```
    /** The socket API (Chapter 15): one thread per client, all sharing one controller. */
    static final class SocketApi {
        private final Controller controller;
        private final ExecutorService workers = Executors.newCachedThreadPool();
        private ServerSocket listener;

        SocketApi(Controller controller) {
            this.controller = controller;
        }

        int start(int port) throws IOException {
            listener = new ServerSocket(port);
            Thread acceptor = new Thread(() -> {
                try {
                    while (true) {
                        Socket c = listener.accept();
                        workers.execute(() -> serve(c));
                    }
                } catch (IOException e) {
                    // listener closed
                }
            });
            acceptor.setDaemon(true);
            acceptor.start();
            return listener.getLocalPort();
        }

        private void serve(Socket c) {
            try (c;
                 BufferedReader in = new BufferedReader(new InputStreamReader(
                         c.getInputStream(), StandardCharsets.UTF_8));
                 PrintWriter out = new PrintWriter(new OutputStreamWriter(
                         c.getOutputStream(), StandardCharsets.UTF_8), true)) {
                out.println(GREETING);
                String line;
                while ((line = in.readLine()) != null && !line.equalsIgnoreCase("QUIT")) {
                    for (String reply : controller.handle(line)) {
                        out.println(reply);
                    }
                    out.println(END);
                }
            } catch (IOException e) {
                // the client vanished: its thread simply ends
            }
        }
```

Listing 16.32 (Continued) *ProfitinaLaundry.java: the controller, the views, the socket API with dot-terminated replies, a client, and the self-test.*

```
        void stop() throws IOException {
            listener.close();
            workers.shutdown();
        }
    }

    /** A client of the socket API: sends one command, reads lines up to the lone dot. */
    static final class ApiClient implements AutoCloseable {
        private final Socket socket;
        private final BufferedReader in;
        private final PrintWriter out;

        ApiClient(int port) throws IOException {
            socket = new Socket("localhost", port);
            socket.setSoTimeout(2000);
            in = new BufferedReader(new InputStreamReader(socket.getInputStream(),
StandardCharsets.UTF_8));
            out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream(),
StandardCharsets.UTF_8), true);
            if (!GREETING.equals(in.readLine())) {
                throw new IOException("not a Profitina Laundry server");
            }
        }

        List<String> call(String command) throws IOException {
            out.println(command);
            List<String> reply = new ArrayList<>();
            String line;
            while (!(line = in.readLine()).equals(END)) {
                reply.add(line);
            }
            return reply;
        }

        /** Tempting but wrong: assumes every reply is exactly one line. */
        String callOneLine(String command) throws IOException {
            out.println(command);
            return in.readLine();
        }

        @Override
        public void close() throws IOException {
            socket.close();
        }
    }

    private static int passed = 0;
```

Listing 16.32 (Continued) *ProfitinaLaundry.java: the controller, the views, the socket API with dot-terminated replies, a client, and the self-test.*

```
    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) throws Exception {
        if (args.length == 2 && args[0].equals("client")) {   // java ProfitinaLaundry
client 5000
            BufferedReader keyboard = new BufferedReader(new InputStreamReader(System.in));
            try (ApiClient client = new ApiClient(Integer.parseInt(args[1]))) {
                String command;
                while ((command = keyboard.readLine()) != null &&
!command.equalsIgnoreCase("QUIT")) {
                    client.call(command).forEach(System.out::println);
                }
            }
            return;
        }
        Path file = (args.length > 1) ? Path.of(args[1]) : Files.createTempFile("laundry",
".csv");
        LaundryCore.Shop shop = new LaundryCore.Shop();
        Controller controller = new Controller(shop, new
LaundryCore.FileOrderRepository(file));
        if (args.length > 0 && args[0].matches("\\d+")) {  // java ProfitinaLaundry 5000
orders.csv
            View console = System.out::println;
            shop.loadFrom(new LaundryCore.FileOrderRepository(file));
            shop.addListener(o -> console.show("[shop] " + describe(o)));
            new SocketApi(controller).start(Integer.parseInt(args[0]));
            console.show("Profitina Laundry listening on port " + args[0] + "; orders in "
+ file);
            Thread.currentThread().join();
        }

        List<String> viewLog = Collections.synchronizedList(new ArrayList<>());
        View recording = viewLog::add;
        shop.addListener(o -> recording.show(describe(o)));

        String[] session = {"PLACE Sari WASH 3.5", "PLACE Budi EXPRESS 2", "PLACE Dewi
DRYCLEAN 1",
                            "PLACE Rudi IRON -2", "ADVANCE L1", "ADVANCE L1", "ADVANCE L9",
                            "LIST", "REVENUE", "FOLD L1"};
        for (String command : session) {
            System.out.println("> " + command);
            controller.handle(command).forEach(r -> System.out.println("  " + r));
        }
        check("view saw every change", 4, viewLog.size());
```

Listing 16.32 (Continued) *ProfitinaLaundry.java: the controller, the views, the socket API with dot-terminated replies, a client, and the self-test.*

```
        SocketApi api = new SocketApi(controller);
        int port = api.start(0);
        ExecutorService crowd = Executors.newFixedThreadPool(5);
        List<Future<List<String>>> futures = new ArrayList<>();
        for (int c = 0; c < 5; c++) {
            String name = "Client" + c;
            futures.add(crowd.submit(() -> {
                List<String> ids = new ArrayList<>();
                try (ApiClient client = new ApiClient(port)) {
                    for (int k = 1; k <= 4; k++) {
                        ids.add(client.call("PLACE " + name + " WASH " + k).get(0).split("
")[1]);
                    }
                }
                return ids;
            }));
        }
        Set<String> ids = new HashSet<>();
        for (Future<List<String>> f : futures) {
            ids.addAll(f.get());
        }
        crowd.shutdown();
        System.out.println("5 clients x 4 orders over the network: " + ids.size() + "
different ids");

        try (ApiClient client = new ApiClient(port)) {
            System.out.println("SAVE    -> " + client.call("SAVE"));
            System.out.println("REVENUE -> " + client.call("REVENUE"));
        }
        String outOfStep;
        try (ApiClient sloppy = new ApiClient(port)) {
            sloppy.callOneLine("LIST");                    // reads only the first of 24
lines
            outOfStep = sloppy.callOneLine("REVENUE");     // ... so this is LIST's second
line
            System.out.println("one-line client asks REVENUE, gets: " + outOfStep);
        }
        api.stop();

        LaundryCore.Shop reopened = new LaundryCore.Shop();
        reopened.loadFrom(new LaundryCore.FileOrderRepository(file));
        LaundryCore.Order next = reopened.place("Nia", "IRON", "1");
        System.out.println("after a restart: " + reopened.all().size() + " orders, next id
" + next.id());
```

Listing 16.32 (Continued) *ProfitinaLaundry.java: the controller, the views, the socket API with dot-terminated replies, a client, and the self-test.*

```
        check("20 different ids", 20, ids.size());
        check("revenue", shop.revenue(), reopened.revenue() - next.price());
        check("numbering continues", "L23", next.id());
        check("the one-line client is out of step", describe(shop.all().get(1)),
outOfStep);
        Files.deleteIfExists(file);
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
> PLACE Sari WASH 3.5
  OK L1 24500
> PLACE Budi EXPRESS 2
  OK L2 28000
> PLACE Dewi DRYCLEAN 1
  ERR unknown service DRYCLEAN
> PLACE Rudi IRON -2
  ERR weight must be in (0, 50] kg
> ADVANCE L1
  OK L1 WASHING
> ADVANCE L1
  OK L1 READY
> ADVANCE L9
  ERR no order L9
> LIST
  L1 Sari WASH 3.5 kg IDR 24,500 READY
  L2 Budi EXPRESS 2.0 kg IDR 28,000 RECEIVED
  OK 2 orders
> REVENUE
  OK 52500
> FOLD L1
  ERR unknown command FOLD
PASS view saw every change
5 clients x 4 orders over the network: 20 different ids
SAVE    -> [OK saved 22 orders]
REVENUE -> [OK 402500]
one-line client asks REVENUE, gets: L2 Budi EXPRESS 2.0 kg IDR 28,000 RECEIVED
after a restart: 23 orders, next id L23
PASS 20 different ids
PASS revenue
PASS numbering continues
PASS the one-line client is out of step
All 5 checks passed.
```

The Python solution. **Listing 16.33** and **Listing 16.34** are the same two layers in Python, with the same three modes on the command line.

Listing 16.33 `laundry_core.py`: the model in Python, with an `RLock` protecting the shop.

```
# laundry_core.py - Problem 16.14 (part 1): the model of Profitina Laundry.
import os
import threading
from abc import ABC, abstractmethod
from dataclasses import dataclass, replace
from enum import Enum


class LaundryError(Exception):
    """A refused request: every caller must decide what it means to it."""


class PriceRule(ABC):
    """Anything that can price a weight (Chapter 6: an interface)."""

    @abstractmethod
    def price_for(self, kg):
        ...


class Service(PriceRule):
    """The services (Chapter 7: an abstract class and a small hierarchy)."""
    code = "?"

    @abstractmethod
    def rate_per_kg(self):
        ...

    def price_for(self, kg):
        return round(self.rate_per_kg() * kg)


class Wash(Service):
    code = "WASH"

    def rate_per_kg(self):
        return 7000


class Iron(Service):
    code = "IRON"

    def rate_per_kg(self):
        return 5000
```

Listing 16.33 (Continued) *laundry_core.py: the model in Python, with an RLock protecting the shop.*

```
class Express(Wash):                            # an express wash IS a wash
    code = "EXPRESS"

    def rate_per_kg(self):
        return 2 * super().rate_per_kg()        # twice the washing rate


SERVICES = {s.code: s for s in (Wash(), Iron(), Express())}


def service(code):
    s = SERVICES.get(code.upper())
    if s is None:
        raise LaundryError(f"unknown service {code}")
    return s


class Status(Enum):
    """The life of an order, in order."""
    RECEIVED = 1
    WASHING = 2
    READY = 3
    PICKED_UP = 4

    def next(self):
        if self is Status.PICKED_UP:
            raise LaundryError("already picked up")
        return Status(self.value + 1)


@dataclass(frozen=True)
class Order:
    """One order (Chapter 13: immutable; a change makes a new Order)."""
    id: str
    customer: str
    service: str
    kg: float
    price: int
    status: Status

    def advanced(self):
        return replace(self, status=self.status.next())
```

Listing 16.33 (Continued) *laundry_core.py: the model in Python, with an RLock protecting the shop.*

```
    def to_line(self):
        return ";".join((self.id, self.customer, self.service, str(self.kg),
                         str(self.price), self.status.name))

    @staticmethod
    def from_line(line):
        p = line.rstrip("\n").split(";")
        return Order(p[0], p[1], p[2], float(p[3]), int(p[4]), Status[p[5]])


class OrderRepository(ABC):
    """Where orders are kept between runs (Chapter 5: files, behind an interface)."""

    @abstractmethod
    def save_all(self, orders):
        ...

    @abstractmethod
    def load_all(self):
        ...


class FileOrderRepository(OrderRepository):
    def __init__(self, file_name):
        self._file_name = file_name

    def save_all(self, orders):
        with open(self._file_name, "w", encoding="utf-8") as f:
            for o in orders:
                f.write(o.to_line() + "\n")

    def load_all(self):
        if not os.path.exists(self._file_name):
            return []
        with open(self._file_name, encoding="utf-8") as f:
            return [Order.from_line(line) for line in f if line.strip()]


class Shop:
    """The model (Chapter 11). Every public method holds the lock: many clients share one
shop."""
```

Listing 16.33 (Continued) *laundry_core.py: the model in Python, with an RLock protecting the shop.*

```
    def __init__(self):
        self._orders = {}                       # a dict keeps arrival order
        self._listeners = []
        self._next_number = 1
        self._lock = threading.RLock()

    def add_listener(self, listener):
        """Observers (the V of MVC) are told about every change."""
        self._listeners.append(listener)

    def _tell(self, order):
        for listener in list(self._listeners):
            listener(order)

    def place(self, customer, service_code, kg_text):
        with self._lock:
            if not customer.strip() or ";" in customer:
                raise LaundryError("customer name is missing or contains ';'")
            s = service(service_code)
            try:
                kg = float(kg_text)
            except ValueError:
                raise LaundryError(f"weight is not a number: {kg_text}") from None
            if not 0 < kg <= 50:
                raise LaundryError("weight must be in (0, 50] kg")
            o = Order(f"L{self._next_number}", customer, s.code, kg, s.price_for(kg),
Status.RECEIVED)
            self._next_number += 1
            self._orders[o.id] = o
            self._tell(o)
            return o

    def advance(self, order_id):
        with self._lock:
            old = self._orders.get(order_id)
            if old is None:
                raise LaundryError(f"no order {order_id}")
            changed = old.advanced()
            self._orders[order_id] = changed    # replace, never modify
            self._tell(changed)
            return changed

    def all(self):
        with self._lock:
            return tuple(self._orders.values())     # a snapshot the caller cannot change
```

Listing 16.33 (Continued) *laundry_core.py: the model in Python, with an RLock protecting the shop.*

```
    def revenue(self):
        with self._lock:
            return sum(o.price for o in self._orders.values())

    def save_to(self, repo):
        with self._lock:
            repo.save_all(self._orders.values())

    def load_from(self, repo):
        with self._lock:
            self._orders.clear()
            for o in repo.load_all():
                self._orders[o.id] = o
                self._next_number = max(self._next_number, int(o.id[1:]) + 1)
```

Listing 16.34 `profitina_laundry.py`: the controller, the socket API, the client and the tests in Python.

```
# profitina_laundry.py - Problem 16.14 (part 2): controller, views and a socket API around
laundry_core.
import os
import socket
import sys
import tempfile
import threading
import unittest
from concurrent.futures import ThreadPoolExecutor
from laundry_core import FileOrderRepository, LaundryError, Shop

GREETING = "PROFITINA-LAUNDRY/1 READY"
END = "."                                       # a line with only a dot ends every reply


def describe(o):
    return f"{o.id} {o.customer} {o.service} {o.kg:.1f} kg IDR {o.price:,} {o.status.name}"


class Controller:
    """The C of MVC: turns one command line into calls on the model and lines of reply."""

    def __init__(self, shop, repo):
        self._shop = shop
        self._repo = repo

    def handle(self, command_line):
        w = command_line.split()
        try:
            command = w[0].upper() if w else ""
            if command == "PLACE":
                self._need(w, 4, "PLACE <customer> <service> <kg>")
                o = self._shop.place(w[1], w[2], w[3])
                return [f"OK {o.id} {o.price}"]
            if command == "ADVANCE":
                self._need(w, 2, "ADVANCE <id>")
                return [f"OK {w[1]} {self._shop.advance(w[1]).status.name}"]
            if command == "LIST":
                lines = [describe(o) for o in self._shop.all()]
                return lines + [f"OK {len(lines)} orders"]
            if command == "REVENUE":
                return [f"OK {self._shop.revenue()}"]
            if command == "SAVE":
                self._shop.save_to(self._repo)
                return [f"OK saved {len(self._shop.all())} orders"]
            return [f"ERR unknown command {w[0] if w else ''}"]
        except LaundryError as e:
```

Listing 16.34 (Continued) *profitina_laundry.py: the controller, the socket API, the client and the tests in Python.*

```
            return [f"ERR {e}"]                # a refused request, not a crash
        except OSError as e:
            return [f"ERR storage failed: {type(e).__name__}"]

    @staticmethod
    def _need(w, n, usage):
        if len(w) != n:
            raise LaundryError(f"usage: {usage}")


class SocketApi:
    """The socket API (Chapter 15): one thread per client, all sharing one controller."""

    def __init__(self, controller):
        self._controller = controller

    def start(self, port):
        self._listener = socket.create_server(("localhost", port))
        threading.Thread(target=self._accept_loop, daemon=True).start()
        return self._listener.getsockname()[1]

    def _accept_loop(self):
        try:
            while True:
                c, _ = self._listener.accept()
                threading.Thread(target=self._serve, args=(c,), daemon=True).start()
        except OSError:
            pass                               # listener closed

    def _serve(self, c):
        try:
            with c, c.makefile("r", encoding="utf-8") as reader, \
                    c.makefile("w", encoding="utf-8") as writer:
                writer.write(GREETING + "\n")
                writer.flush()
                for line in reader:
                    if line.strip().upper() == "QUIT":
                        return
                    for reply in self._controller.handle(line):
                        writer.write(reply + "\n")
                    writer.write(END + "\n")
                    writer.flush()
        except OSError:
            pass                               # the client vanished: its thread simply
ends
```

Listing 16.34 (Continued) *profitina_laundry.py: the controller, the socket API, the client and the tests in Python.*

```
    def stop(self):
        try:
            self._listener.shutdown(socket.SHUT_RDWR)
        except OSError:
            pass
        self._listener.close()


class ApiClient:
    """A client of the socket API: sends one command, reads lines up to the lone dot."""

    def __init__(self, port):
        self._socket = socket.create_connection(("localhost", port), timeout=2)
        self._reader = self._socket.makefile("r", encoding="utf-8")
        self._writer = self._socket.makefile("w", encoding="utf-8")
        if self._reader.readline().rstrip("\n") != GREETING:
            raise OSError("not a Profitina Laundry server")

    def _send(self, command):
        self._writer.write(command + "\n")
        self._writer.flush()

    def call(self, command):
        self._send(command)
        reply = []
        while (line := self._reader.readline().rstrip("\n")) != END:
            reply.append(line)
        return reply

    def call_one_line(self, command):
        """Tempting but wrong: assumes every reply is exactly one line."""
        self._send(command)
        return self._reader.readline().rstrip("\n")

    def close(self):
        self._socket.close()

    def __enter__(self):
        return self

    def __exit__(self, *exc):
        self.close()
```

Listing 16.34 (Continued) *profitina_laundry.py: the controller, the socket API, the client and the tests in Python.*

```
SESSION = ["PLACE Sari WASH 3.5", "PLACE Budi EXPRESS 2", "PLACE Dewi DRYCLEAN 1",
           "PLACE Rudi IRON -2", "ADVANCE L1", "ADVANCE L1", "ADVANCE L9",
           "LIST", "REVENUE", "FOLD L1"]


def crowd(port):
    """5 clients x 4 orders over the network, at the same time; returns all ids."""
    def one(name):
        with ApiClient(port) as client:
            return [client.call(f"PLACE {name} WASH {k}")[0].split()[1] for k in range(1,
5)]
    with ThreadPoolExecutor(max_workers=5) as pool:
        return {i for ids in pool.map(one, [f"Client{c}" for c in range(5)]) for i in ids}


class TestProfitinaLaundry(unittest.TestCase):
    def setUp(self):
        fd, self.file = tempfile.mkstemp(suffix=".csv")
        os.close(fd)
        self.shop = Shop()
        self.controller = Controller(self.shop, FileOrderRepository(self.file))

    def tearDown(self):
        os.remove(self.file)

    def test_session(self):
        replies = [self.controller.handle(c) for c in SESSION]
        self.assertEqual(["OK L1 24500"], replies[0])
        self.assertEqual(["ERR unknown service DRYCLEAN"], replies[2])
        self.assertEqual(["OK L1 READY"], replies[5])
        self.assertEqual(["OK 52500"], replies[8])

    def test_network_save_and_restart(self):
        api = SocketApi(self.controller)
        port = api.start(0)
        self.assertEqual(20, len(crowd(port)))
        with ApiClient(port) as client:
            self.assertEqual(["OK saved 20 orders"], client.call("SAVE"))
        api.stop()
        reopened = Shop()
        reopened.load_from(FileOrderRepository(self.file))
        self.assertEqual(self.shop.revenue(), reopened.revenue())
        self.assertEqual("L21", reopened.place("Nia", "IRON", "1").id)
```

Listing 16.34 (Continued) *profitina_laundry.py: the controller, the socket API, the client and the tests in Python.*

```
    def test_one_line_client_is_out_of_step(self):
        self.controller.handle("PLACE Sari WASH 3.5")
        self.controller.handle("PLACE Budi EXPRESS 2")
        api = SocketApi(self.controller)
        with ApiClient(api.start(0)) as sloppy:
            sloppy.call_one_line("LIST")
            self.assertEqual(describe(self.shop.all()[1]), sloppy.call_one_line("REVENUE"))
        api.stop()


def serve(port, file_name):                    # python profitina_laundry.py 5000
orders.csv
    shop = Shop()
    shop.load_from(FileOrderRepository(file_name))
    shop.add_listener(lambda o: print("[shop]", describe(o), flush=True))
    SocketApi(Controller(shop, FileOrderRepository(file_name))).start(port)
    print(f"Profitina Laundry listening on port {port}; orders in {file_name}", flush=True)
    threading.Event().wait()


def client(port):                              # python profitina_laundry.py client 5000
    with ApiClient(port) as c:
        for command in sys.stdin:
            if command.strip().upper() == "QUIT":
                break
            print("\n".join(c.call(command.strip())))


def demo(folder):
    """The same run as the Java main method: a session, the network, a restart."""
    file_name = os.path.join(folder, "orders.csv")
    shop, view_log = Shop(), []
    shop.add_listener(lambda o: view_log.append(describe(o)))
    controller = Controller(shop, FileOrderRepository(file_name))
    for command in SESSION:
        print(">", command)
        for reply in controller.handle(command):
            print(" ", reply)
    print("view saw", len(view_log), "changes")
    api = SocketApi(controller)
    port = api.start(0)
    print("5 clients x 4 orders over the network:", len(crowd(port)), "different ids")
    with ApiClient(port) as c:
        print("SAVE    ->", c.call("SAVE"))
        print("REVENUE ->", c.call("REVENUE"))
    with ApiClient(port) as sloppy:
        sloppy.call_one_line("LIST")                    # reads only the first of 24 lines
```

Listing 16.34 (Continued) *profitina_laundry.py: the controller, the socket API, the client and the tests in Python.*

```
        print("one-line client asks REVENUE, gets:", sloppy.call_one_line("REVENUE"))
    api.stop()
    reopened = Shop()
    reopened.load_from(FileOrderRepository(file_name))
    nxt = reopened.place("Nia", "IRON", "1")
    print(f"after a restart: {len(reopened.all())} orders, next id {nxt.id}")
    os.remove(file_name)


if __name__ == "__main__":
    if len(sys.argv) == 3 and sys.argv[1] == "client":
        client(int(sys.argv[2]))
    elif len(sys.argv) >= 2 and sys.argv[1].isdigit():
        serve(int(sys.argv[1]), sys.argv[2] if len(sys.argv) > 2 else "orders.csv")
    else:
        demo(tempfile.mkdtemp())
        sys.stdout.flush()
        unittest.main(verbosity=2)
```

```
> PLACE Sari WASH 3.5
  OK L1 24500
> PLACE Budi EXPRESS 2
  OK L2 28000
> PLACE Dewi DRYCLEAN 1
  ERR unknown service DRYCLEAN
> PLACE Rudi IRON -2
  ERR weight must be in (0, 50] kg
> ADVANCE L1
  OK L1 WASHING
> ADVANCE L1
  OK L1 READY
> ADVANCE L9
  ERR no order L9
> LIST
  L1 Sari WASH 3.5 kg IDR 24,500 READY
  L2 Budi EXPRESS 2.0 kg IDR 28,000 RECEIVED
  OK 2 orders
> REVENUE
  OK 52500
> FOLD L1
  ERR unknown command FOLD
view saw 4 changes
5 clients x 4 orders over the network: 20 different ids
SAVE    -> ['OK saved 22 orders']
REVENUE -> ['OK 402500']
one-line client asks REVENUE, gets: L2 Budi EXPRESS 2.0 kg IDR 28,000 RECEIVED
after a restart: 23 orders, next id L23
test_network_save_and_restart (__main__.TestProfitinaLaundry.test_network_save_and_restart)
... ok
test_one_line_client_is_out_of_step
(__main__.TestProfitinaLaundry.test_one_line_client_is_out_of_step) ... ok
test_session (__main__.TestProfitinaLaundry.test_session) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.018s

OK
```

Run as a server and used from a second terminal, the Java program behaves as follows. The first terminal is the server's console view, where the shop's listener prints every change:

```
Profitina Laundry listening on port 5000; orders in orders.csv
[shop] L1 Sari WASH 3.5 kg IDR 24,500 RECEIVED
[shop] L2 Budi IRON 2.0 kg IDR 10,000 RECEIVED
[shop] L1 Sari WASH 3.5 kg IDR 24,500 WASHING
```

and the second terminal is a counter, typing commands (the lines typed by the user are those beginning with a command word):

```
PLACE Sari WASH 3.5
OK L1 24500
PLACE Budi IRON 2
OK L2 10000
ADVANCE L1
OK L1 WASHING
LIST
L1 Sari WASH 3.5 kg IDR 24,500 WASHING
L2 Budi IRON 2.0 kg IDR 10,000 RECEIVED
OK 2 orders
SAVE
OK saved 2 orders
QUIT
```

After `SAVE`, the file `orders.csv` contains one line per order, `L1;Sari;WASH;3.5;24500;WASHING` and `L2;Budi;IRON;2.0;10000;RECEIVED`, which the server reads back when it is started again.

Line by line. We walk through the decisions layer by layer, from the inside out.

The services. `PriceRule` is an interface with one method, `priceFor(kg)`. The abstract class `Service` implements it once, in terms of two abstract methods, `code()` and `ratePerKg()`; this is the *template method* pattern of Chapter 8. `Wash` and `Iron` supply the numbers. `Express extends Wash`, because an express wash *is* a wash, and its `ratePerKg()` returns `2 * super.ratePerKg()`: if the washing rate changes, express follows automatically. `SERVICES` maps codes to the three objects, and `service(code)` turns an unknown code into a checked `LaundryException`.

The orders. `Status` is an enum whose `next()` returns the following constant (`values()[ordinal() + 1]`) and refuses to go beyond `PICKED_UP`. `Order` is a record, so it is immutable; `advanced()` returns a *new* order with the next status. `toLine` and `fromLine` convert an order to a line of the file and back; the customer name may not contain `;`, which `place` checks, because the file uses `;` as its separator.

The repository. `OrderRepository` is the storage contract, like `JokeStore` in Problem 16.1: the shop depends on the interface, and `FileOrderRepository` implements it with `Files.write` and `Files.readAllLines`. A database implementation could replace it without any change in the shop.

The shop. The orders live in a `LinkedHashMap`, a hash map that also remembers insertion order, so `LIST` shows orders in the order they arrived. Every public method that touches the orders is `synchronized`, because the socket API calls the one shop from many threads; the check in the output, “20 different ids” from 5 clients placing 4 orders each at the same time, is the test of that decision. Without `synchronized`, two threads could read the same `nextNumber` and create two orders called, say, L7: the lost update of Problem 16.11 in a new place. `advance` *replaces* the order in the map with the new object, so a snapshot returned earlier by `all()` (`List.copyOf`, unmodifiable) never changes under the caller’s feet. The listeners are kept in a `CopyOnWriteArrayList`, a list that is safe to iterate while another thread adds to it. `loadFrom` restores the orders and sets `nextNumber` to one more than the largest number found, which is why the order placed after the restart is L23.

The controller. `handle` splits the command, checks the number of words with `need`, calls the model, and converts every `LaundryException` into a line starting with `ERR`. That conversion is the

most important line of the controller: a refused request produces a polite reply, never a dead connection thread. The session in the output exercises every command and every kind of refusal: an unknown service, a negative weight, a missing order and an unknown command. The revenue after the session is $24{,}500 + 28{,}000 = 52{,}500$.

The views. `View` has one method, `show(String)`, so a lambda or a method reference is a view: `System.out::println` for the server's console, `viewLog::add` for the test. The shop does not know which views exist; it only calls its listeners. That is the observer relationship at the heart of MVC (Chapter 11).

The socket API. `SocketApi.serve` is the server loop of Problem 16.3 with the framing rule added: after the reply lines of each command it writes a lone `.`. `ApiClient.call` reads lines until that dot. In the output, five clients place twenty orders concurrently; `SAVE` and `REVENUE` then report 22 orders and a revenue of $52{,}500 + 5 \times 7000 \times (1 + 2 + 3 + 4) = 402{,}500$. A fresh `Shop` loaded from the file places one more order and receives the identifier L23, and its revenue minus that new order equals the old shop's revenue: the file round trip lost nothing.

The Python version makes the same decisions with Python's tools: abstract base classes for the interfaces, a class attribute `code` in each service, a frozen dataclass with `dataclasses.replace` for `advanced()`, a dictionary (which keeps insertion order) for the orders, and an `RLock`, a lock that the same thread may acquire again, so that a listener may safely call back into the shop.

A tempting wrong solution. `callOneLine` is the client that worked perfectly for the one-line protocol of Problem 16.13 and was reused unchanged. After `LIST`, it reads only the first of the twenty-four reply lines. The next request, `REVENUE`, then reads the *second* line of the old `LIST` reply, the order L2, instead of the revenue, and every later reply is out of step as well. Nothing crashes; the client simply believes wrong answers. The output shows it: `one-line client asks REVENUE, gets: L2 Budi EXPRESS 2.0 kg IDR 28,000 RECEIVED`. When replies can have several lines, the protocol needs an explicit end marker, and the client must read up to it.

> **CHECK YOURSELF.** A new service, dry cleaning, costs 20,000 per kilogram. Which classes change, and what does `PLACE Ayu DRYCLEAN 1.5` reply in a fresh shop?
>
> *Answer.* One new class, `DryClean extends Service` with `code()` returning `"DRYCLEAN"` and `ratePerKg()` returning 20000, and one new entry in `SERVICES` (Python: add `DryClean()` to the tuple). Nothing else changes: the controller, the shop, the file format and the socket API work with any `Service`. The reply is `OK L1 30000`, since $20{,}000 \times 1.5 = 30{,}000$.

16.11 The Problems of This Chapter Compared

Table 16.11 lays the fourteen problems side by side. As in Chapter 8, read down the last column: almost every wrong solution passes the obvious test with one client, one thread or one line, and fails only under the condition the problem was about.

Table 16.11 The fourteen problems of the chapter: what each practices and the single case its tempting wrong solution gets wrong.

Problem	Topic	Main idea	What the tempting wrong solution gets wrong
16.1	contracts	interface plus contract test	numbering from 1: fails `get(0)` and `get(size())`
16.2	protocol	state machine, one line per turn	no state: answers any "… who?"
16.3	server	one thread per client	one at a time: the second client starves
16.4	testing	each side tests the other with stubs	a weak test passes a mutant client
16.5	concurrency	start gate, oracle, deadline	one shared protocol object
16.6	failures	time limits; end of stream is normal	`null` or `""` not checked; crash or busy loop
16.7	GUI	network off the UI thread; MVC	network in the button handler: frozen window
16.8	documentation	measured, repeatable criteria	one lucky measurement written as a fact
16.9	iterators	a new iterator per loop; fail fast	the collection is its own iterator
16.10	value objects	immutable; `equals` with `hashCode`	no `hashCode`; a mutable key becomes a ghost
16.11	races	lock, `synchronized` or atomic	`count++`, even `volatile`; "the test passed"
16.12	producer–consumer	`while` around `wait`; poison pills	`if` around `wait`
16.13	protocols	pure function; flush every request	no flush: the request never leaves
16.14	the whole book	layers, MVC, framing	one-line reads of multi-line replies

16.12 The Whole Book in One Table

Sixteen chapters ago, this book began with a program that printed one line. **Table 16.12** is the road from there to the capstone, one row per chapter, with the question each chapter answered and the place where its answer is used in this chapter.

Table 16.12 The whole book in one table: each chapter's question and where its answer is used in this chapter.

Chapter	The question it answered	Used in this chapter by
1	How do I write and run a program in Java and in Python?	every listing; the README of the chapter's code
2	How do I edit, run and debug in an environment?	running two terminals; debugging threads
3	What are classes, objects, strings, types and loops?	every class; `describe`, `String.format`
4	How do I go from a specification to a tested class?	the problem route of this chapter
5	How do I use arrays, lists, input, exceptions and files?	`PickupSchedule`, `FileJokeStore`, `FileOrderRepository`
6	How do I document, test and write interfaces?	`JokeStore`, `View`, contract tests
7	How do inheritance, polymorphism and access control work?	`Service`, `Wash`, `Express`; private fields
8	How do I design several cooperating classes?	the ten-step method of Section 16.2
9	How do collections and generics work?	maps, lists, sets, queues, `Future<T>`
10	How do I build a window and handle events?	`KnockKnockGui`, the event dispatch thread
11	How do I separate model, view and controller?	Problems 16.7 and 16.14
12	How do I combine collections, windows and MVC in one program?	the capstone's layers
13	How do iterators and immutable objects work?	Problems 16.9 and 16.10; `Order`, `Joke`
14	How do threads share data safely?	Problems 16.5, 16.11, 16.12; the shop
15	How do programs talk over a network?	Problems 16.3 to 16.8, 16.13, 16.14

Read down the middle column and you will see that the questions changed character halfway through. The first chapters asked "how do I write this?"; the later ones asked "how do these parts work together, and what happens when several things happen at once?" Object-oriented programming is, in the end, the craft of the second kind of question: dividing a system into objects with clear responsibilities and clear contracts, so that each part can be understood, tested and changed alone.

16.13 Where to Go Next

You can now build a complete application in two languages. Three kinds of knowledge will make the next application better, faster to build and easier for others to join: design patterns, build tools and frameworks.

16.13.1 Design patterns

A *design pattern* is a named, proven solution to a problem that recurs in object-oriented design [6]. You have already used several without the name, and **Table 16.13** gives them their names. Knowing the names lets a team say in one word what would otherwise take a paragraph ("the shop is the subject, the views are observers"). The classic catalogue is Gamma et al. [6]; a gentler introduction is Freeman and Robson [7].

Table 16.13 Design patterns used in this book, by name.

Pattern	Its purpose	Where you met it
Iterator	walk a collection without knowing its structure	Problem 16.9
Observer	tell many objects about changes in one	the shop's listeners, Chapters 10–11
Model–View–Controller	separate data, display and input handling	Chapter 11, Problems 16.7, 16.14
Strategy	make an algorithm a replaceable object	`PriceRule`, the user function of the client
Template method	fix the steps, let subclasses fill some in	`Service.priceFor`, Chapter 8
Factory method	create objects without naming their class	`JokeStore.of`, `Money.of`
Repository	hide storage behind a collection-like interface	`JokeStore`, `OrderRepository`
Producer–consumer	decouple producers and consumers with a buffer	Problem 16.12
State	make an object's behavior depend on its state	`KnockKnockProtocol`

A word of caution comes with patterns: they are solutions to problems, not goals. A class that uses three patterns to add two numbers is worse than one that simply adds them. Reach for a pattern when you recognize its problem.

16.13.2 Build tools and environments

The programs of this book were compiled with `javac` and run with `java` or `python`, one folder at a time. Real projects depend on libraries (JUnit, a database driver, a web framework), and managing those by hand quickly becomes impossible. *Build tools* describe a project in one file and do the rest.

In Java, the two standard tools are **Maven** [8] and **Gradle** [9]. A Maven project is described by a file `pom.xml` that names the project and its dependencies; the command `mvn test` then downloads the dependencies, compiles the code in `src/main/java`, compiles and runs the tests in `src/test/java`, and prints a report. Gradle does the same with a file `build.gradle` (or `build.gradle.kts`) and the command `gradle test`. Both follow the same folder convention, which every Java editor understands, and both can package the program into a single JAR file that runs with `java -jar`.

In Python, the equivalents are a *virtual environment* and **pip** [10]. A virtual environment is a private folder of installed packages for one project, so that two projects can use different versions of the same library:

```
python -m venv .venv                  create the environment (once)
.venv\Scripts\activate                activate it on Windows
source .venv/bin/activate             activate it on macOS and Linux
pip install pytest                    install a package into it
pip freeze > requirements.txt         record the exact versions
pip install -r requirements.txt       recreate the same environment elsewhere
```

A modern Python project describes itself in `pyproject.toml`, and `pytest`, the most widely used test runner, runs `unittest` tests unchanged as well as its own shorter style. Profitina itself pins every dependency in a lock file, for the reason `requirements.txt` exists: a program that works today must still work after the next installation.

Two more tools belong to every project from its first day: **Git**, for version control, which records every change and lets a team work on the same files; and *continuous integration*, a server that runs the whole test suite on every change, which is the automatic form of Step 6 of Section 16.2.

16.13.3 Frameworks

A *framework* is a library that calls your code rather than the other way round: you write the classes, and the framework decides when to create them and call their methods. The socket servers of this chapter are exactly what frameworks do for you, with HTTP instead of our line protocols, and with far more care about security and performance. In Java, **Spring Boot** [11] is the most widely used framework for web services: in the 2025 Stack Overflow Developer Survey it is the only Java framework among the listed web frameworks, used by 14.7% of the respondents to that question [12]; and **JavaFX** is the modern successor of Swing for desktop windows. In Python, **FastAPI** [13] builds web APIs from ordinary functions with type hints, **Django** is a complete web framework with a database layer and an administration site, and **Flask** is a small one; the same survey counts 14.8%, 12.6% and 14.4% of respondents using them [12]. Profitina, whose shape is described in Section 16.14, is built with Python and FastAPI.

Everything in this book carries over. A Spring controller is the `Controller` of Problem 16.14 with annotations; a FastAPI endpoint is the pure function `reply` of Problem 16.13 with HTTP around it; a JPA entity or a Django model is a class like `Order` with its repository. Learning a framework after this book is learning where the framework puts the pieces you already know.

> **A READING PATH.** For Java: Bloch, *Effective Java* [14], then Goetz et al., *Java Concurrency in Practice* [15], then the Spring Boot guides. For Python: Ramalho, *Fluent Python* [16], then the FastAPI tutorial. For both: Gamma et al. [6] for patterns, Fowler's *Refactoring* [17] for improving code that works, and Brooks, *The Mythical Man-Month* [18], for why adding people to a late project makes it later.

16.14 Profitina in Practice

Profitina, an AI-visibility (GEO) platform in production at app.profitina.com, measures how ready a brand's website is to be cited by AI answer engines and then helps fix what it finds. It is a Python and FastAPI application, served by Uvicorn, and almost every problem of this chapter has a counterpart in its public shape.

Contracts between parts. Profitina's scanner examines a site through a set of *signals*, each scored and explained independently behind one common interface, which is Problem 16.1's `JokeStore` idea at product scale: a new signal is a new implementation, checked by the same contract tests. And the whole automated suite, well over a thousand tests in Python plus JavaScript guards for the interface, must pass before any release: the test plans of Problems 16.4 to 16.6, made into a gate.

A real-time server with many clients. The community features of Profitina push new posts to every open browser through *Server-Sent Events*: long-lived connections on which the server writes a line whenever something happens. It is the server of Problem 16.3 with HTTP around it, and it faces the same questions: every connection has its own state, shared data must be immutable or protected, and a client that disappears must be noticed and cleaned up. The dashboard gathers everything a user needs in one request, `GET /api/me/dashboard`, which is the request–reply design of Problem 16.13: one request, one complete reply.

Failure handled honestly. When a user posts, the composer clears at once and the post appears immediately, and if sending fails the text is restored: the client-side failure detection of Problem 16.6, expressed in the interface. The deployment works the same way on the server side: a release is reported as successful only after the deployed version answers its health check at `/api/health`, a request whose only purpose is to find out whether the server is really up. And when no local AI model is available, the application *says so* instead of silently serving a template, the principle of Problem 16.6 that the user must always be told what really happened.

Personal data, handled in one place. Profitina's account features are built for Indonesia's personal data protection law, UU PDP 27/2022, with explicit consent, data export and full account deletion. Rules like these can only be kept when personal data is changed in one controlled place, as the synchronized `Shop` of Problem 16.14 is the one place where orders change.

As always, the product advises and its users decide: a visibility score is a measurement with an explanation, not a guarantee. The runnable version of every idea above is the Profitina Laundry code of this chapter.

16.15 Common Mistakes, and How to Avoid Them

Agreeing on contracts only in words. Write each contract as an interface or a protocol table, and a contract test that every implementation runs (Problems 16.1, 16.2).

A protocol without an answer for every input. A flow chart shows the happy path; a state table with an "anything else" row in every state shows the rest (Problem 16.2).

No framing rule. Decide how the receiver knows where a message ends: one line per message, or a terminator line for multi-line replies (Problems 16.2, 16.14).

Serving clients one at a time. A server that handles a client inside its accept loop starves everyone else; give each connection its own thread or task (Problem 16.3).

Sharing per-connection state. A conversation's state belongs to the connection. Share only immutable or thread-safe objects (Problems 16.3, 16.5).

Tests that cannot fail. Check exact results, not "something happened"; break the code on purpose and make sure a test notices (Problem 16.4).

Blocking calls without a time limit. A connect, a read or a `join` without a limit can hang a program and its tests forever. Use timeouts and deadlines (Problems 16.5, 16.6).

Forgetting the end of the stream. `readLine()` returns `null` in Java and `readline()` returns `""` in Python when the other side closes; test for it on every read (Problem 16.6).

Network code on the user-interface thread. Keep the event dispatch thread and the Tk main loop free; do all waiting on another thread and post view updates back (Problem 16.7).

Performance claims from one run. Warm up, repeat, report a median and a maximum, and state the condition (Problem 16.8).

Returning `this` from `iterator()`. Every loop needs its own iterator; and a Python generator object can be looped over only once (Problem 16.9).

`equals` without `hashCode`, and mutable keys. Override both from the same fields, and make keys immutable (Problem 16.10).

Believing that a passing test proves there is no race. Protect every variable that several threads read and write, and widen the window on purpose to test the fix (Problem 16.11).

`if` around `wait`. Always `while`; use `notifyAll`; one poison pill per consumer (Problem 16.12).

Forgetting to flush. A buffered writer keeps a request to itself; flush after every request, or use autoflush (Problem 16.13).

16.16 Chapter Summary and a Closing Word

- A team project is approached in ten steps (Section 16.2): a small goal, user stories, parts with owners, contracts first, the protocol as a state machine, cross tests with stubs, a walking skeleton, deliberate concurrency and failure handling, measurement and documentation, and a report written as the work proceeds.
- Contracts are interfaces or protocols written before the code, each with a test that any implementation must pass (16.1). A protocol is specified by a state machine, Equation (16.1), and a transition table with a row for every input (16.2).
- A multi-client server gives each connection its own thread and its own state; shared objects are immutable or thread-safe (16.3). Each side tests the other through the contract, with stubs (16.4); the running system is tested with many clients released by a start gate and checked against an oracle (16.5).
- Every blocking call has a time limit; the end of the stream is a normal result; every failure becomes a clear message for the user or a log entry on the server, and the server keeps serving (16.6). A window never waits for the network (16.7). Performance criteria are measured, repeated and stated with their condition, Equation (16.2) and Equation (16.3) (16.8).
- An iterable hands out a new iterator for every loop, and its iterator can fail fast, Equation (16.4) (16.9). A value object is immutable and its `hashCode` agrees with `equals` (16.10).
- A race loses updates, Equation (16.5) and Equation (16.6); `synchronized`, locks and atomic variables cure it, and a passing test does not prove its absence (16.11). A bounded buffer waits in `while` loops and stops its consumers with one pill each (16.12).
- A request–reply protocol is best written as a pure function and must flush every request (16.13). The capstone combines every chapter of the book into one layered, tested, networked application (16.14).

A closing word. When you opened this book, a class was a word you had perhaps heard and an object was something on your desk. You have now written classes that promise things to each other and keep those promises, objects that talk across a network, threads that share work without spoiling it, and windows that stay alive while the world is slow. More important than any of these programs is the habit of mind behind them: to ask, of every part of a system, what it knows, what it does, whom it trusts, and what it promises; to write that promise down; and to test it before believing it. That habit does not belong to Java or to Python. It will serve you in the languages and frameworks that do not exist yet, in teams you have not yet joined, on problems nobody has yet posed. Build something you care about with it. Build it with others, write down what you promise each other, and test it. Then build the next thing a little better. That is what programmers do, and you are one now.

16.17 Exercises

Exercises 16.1, 16.2, 16.3, 16.6, 16.8 and 16.10 ask for code; their solutions are programs in Section 16.18. The others ask for reasoning and are answered in words.

1. **(Countdown.)** Write a class `Countdown` that is `Iterable<Integer>` and yields $n, n-1, \ldots, 1$, like the display of a lift going down. In Python, write it as a generator function. What does a loop over `Countdown(0)` do?
2. **(Removing during a loop.)** The list of pickup hours is `[9, 14, 11, 17, 9]`. Remove every hour before 12 in two correct ways in Java, and one in Python. Why does a for-each loop that calls `list.remove` fail?
3. **(Splitting money.)** Write `split(Money m, int n)` that divides an amount into n parts which differ by at most one unit and add up exactly to the original. What is `split(IDR 10,000, 3)`?
4. **(A race by hand.)** Two threads each execute `count++` three times on an unsynchronized counter that starts at 0. What are the largest and the smallest possible final values? Describe an interleaving that gives the smallest.
5. **(Deadlock.)** Two washing machines each hold a queue of orders protected by its own lock. To move an order from machine A to machine B, a thread locks A's queue, then B's; to move one from B to A, another thread locks B's, then A's. Show how the two threads can deadlock, and give a rule that prevents it.
6. **(A TOTAL request.)** Extend LAUNDRY/1 with `TOTAL <service> <kg> [<service> <kg> ...]`, which returns `OK` and the sum of the prices, or the first `ERR` reply. Reuse `reply`. What does `TOTAL WASH 3 IRON 2` return?
7. **(Heartbeats.)** Design a heartbeat for the knock-knock protocol so that the server notices within 15 seconds that a client's cable was unplugged. Which messages are added, who sends them, and what changes in **Table 16.5**?
8. **(A database store.)** Write `SqliteJokeStore` in Python with the `sqlite3` module and show that it keeps the contract of Problem 16.1. What would a Java `JdbcJokeStore` need that the JDK does not include?
9. **(Troubleshooting.)** Write troubleshooting entries, in the style of **Table 16.8**, for "the window says *Connected* but nothing appears" and for "the server's console shows `Too many open files`".
10. **(An observer over the network.)** Add to the capstone a listener that counts how many orders have reached `READY`, and test it through the socket API: place two orders, advance one to `READY` and the other to `PICKED_UP`, and check the count.
11. **(Test design.)** The GUI member of Problem 16.1 uses the client member's `converse`. List five tests the GUI member should write for it, each with its input and its expected result.
12. **(Where next.)** Describe, in the folder layout of Maven, where `LaundryCore.java`, `ProfitinaLaundry.java` and a JUnit test class for `Shop` would go, and which command runs the tests.

16.18 Solutions to the Exercises

Exercises 1, 2, 3, 6 and 10. **Listing 16.35** contains the Java solutions, each checked against the values worked out below; compile it together with the chapter's other files (`javac *.java`), since it uses `Money`, `LaundryProtocol`, `LaundryCore` and `ProfitinaLaundry`.

Listing 16.35 `ExerciseSolutions.java`: `Countdown`, correct removal during a loop, `split`, `TOTAL`, and a `READY` counter tested over the network.

```
// ExerciseSolutions.java - solutions to the programming exercises of Chapter 16.
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.NoSuchElementException;
import java.util.concurrent.atomic.AtomicInteger;

public class ExerciseSolutions {

    /** Exercise 16.1: n, n-1, ..., 1, like the display of a lift going down. */
    static final class Countdown implements Iterable<Integer> {
        private final int from;

        Countdown(int from) {
            this.from = from;
        }

        @Override
        public Iterator<Integer> iterator() {
            return new Iterator<>() {
                private int next = from;

                @Override
                public boolean hasNext() {
                    return next >= 1;
                }

                @Override
                public Integer next() {
                    if (!hasNext()) {
                        throw new NoSuchElementException();
                    }
                    return next--;
                }
            };
        }
    }
```

Listing 16.35 (Continued) *ExerciseSolutions.java: Countdown, correct removal during a loop, split, TOTAL, and a READY counter tested over the network.*

```
    /** Exercise 16.2: removing the pickups before 12 o'clock, correctly, in two ways. */
    static List<Integer> removeMorningWithIterator(List<Integer> hours) {
        List<Integer> copy = new ArrayList<>(hours);
        Iterator<Integer> it = copy.iterator();
        while (it.hasNext()) {
            if (it.next() < 12) {
                it.remove();                        // the iterator's own remove is allowed
            }
        }
        return copy;
    }

    static List<Integer> removeMorningWithRemoveIf(List<Integer> hours) {
        List<Integer> copy = new ArrayList<>(hours);
        copy.removeIf(h -> h < 12);
        return copy;
    }

    /** Exercise 16.3: n parts that differ by at most one unit and add up exactly. */
    static List<Money> split(Money m, int n) {
        if (n <= 0) {
            throw new IllegalArgumentException("n must be positive: " + n);
        }
        long base = m.amount() / n;
        long extra = m.amount() % n;                // the first "extra" parts get one unit
more
        List<Money> parts = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            parts.add(Money.of(base + (i < extra ? 1 : 0), m.currency()));
        }
        return parts;
    }

    /** Exercise 16.6: TOTAL WASH 3 IRON 2 adds up several PRICE replies. */
    static String total(String request) {
        String[] w = request.trim().split("\\s+");
        if (w.length < 3 || w.length % 2 == 0) {
            return "ERR usage: TOTAL <service> <kg> [<service> <kg> ...]";
        }
        long sum = 0;
        for (int i = 1; i < w.length; i += 2) {
            String one = LaundryProtocol.reply("PRICE " + w[i] + " " + w[i + 1]);
            if (one.startsWith("ERR")) {
                return one;                         // the first error decides
            }
            sum += Long.parseLong(one.substring(3));
        }
```

Listing 16.35 (Continued) *ExerciseSolutions.java: Countdown, correct removal during a loop, split, TOTAL, and a READY counter tested over the network.*

```
        return "OK " + sum;
    }

    private static int passed = 0;

    private static void check(String name, Object expected, Object actual) {
        if (!java.util.Objects.equals(expected, actual)) {
            throw new AssertionError(name + ": expected " + expected + " but got " +
actual);
        }
        passed++;
        System.out.println("PASS " + name);
    }

    public static void main(String[] args) throws Exception {
        List<Integer> seen = new ArrayList<>();
        for (int floor : new Countdown(4)) {
            seen.add(floor);
        }
        check("16.1 countdown", List.of(4, 3, 2, 1), seen);
        check("16.1 empty countdown", false, new Countdown(0).iterator().hasNext());

        List<Integer> hours = List.of(9, 14, 11, 17, 9);
        check("16.2 iterator remove", List.of(14, 17), removeMorningWithIterator(hours));
        check("16.2 removeIf", List.of(14, 17), removeMorningWithRemoveIf(hours));

        List<Money> parts = split(Money.rupiah(10_000), 3);
        check("16.3 parts", "[IDR 3,334, IDR 3,333, IDR 3,333]", parts.toString());
        check("16.3 nothing lost", Money.rupiah(10_000),
parts.get(0).plus(parts.get(1)).plus(parts.get(2)));

        check("16.6 total", "OK 31000", total("TOTAL WASH 3 IRON 2"));
        check("16.6 error", "ERR unknown service SILK", total("TOTAL WASH 3 SILK 1"));

        // Exercise 16.10: an observer that counts orders reaching READY, tested over the
network.
        LaundryCore.Shop shop = new LaundryCore.Shop();
        AtomicInteger ready = new AtomicInteger();
        shop.addListener(o -> {
            if (o.status() == LaundryCore.Status.READY) {
                ready.incrementAndGet();
            }
        });
        ProfitinaLaundry.SocketApi api = new ProfitinaLaundry.SocketApi(
                new ProfitinaLaundry.Controller(shop, new LaundryCore.FileOrderRepository(
                        java.nio.file.Path.of("unused.csv"))));
        int port = api.start(0);
        try (ProfitinaLaundry.ApiClient c = new ProfitinaLaundry.ApiClient(port)) {
            c.call("PLACE Sari WASH 3");
```

Listing 16.35 (Continued) *ExerciseSolutions.java: Countdown, correct removal during a loop, split, TOTAL, and a READY counter tested over the network.*

```
            c.call("PLACE Budi IRON 1");
            for (String id : new String[] {"L1", "L1", "L2", "L2", "L2"}) {
                c.call("ADVANCE " + id);            // L1 -> READY; L2 -> READY ->
PICKED_UP
            }
        }
        api.stop();
        check("16.10 two orders reached READY", 2, ready.get());
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
PASS 16.1 countdown
PASS 16.1 empty countdown
PASS 16.2 iterator remove
PASS 16.2 removeIf
PASS 16.3 parts
PASS 16.3 nothing lost
PASS 16.6 total
PASS 16.6 error
PASS 16.10 two orders reached READY
All 9 checks passed.
```

Listing 16.36 contains the Python solutions, including Exercise 8.

Listing 16.36 `exercise_solutions.py`: the same solutions in Python, and the SQLite joke store of Exercise 8.

```
# exercise_solutions.py - solutions to the programming exercises of Chapter 16.
import os
import sqlite3
import unittest
from joke_store import Joke, JokeStore
from joke_stores import contract
from laundry_core import FileOrderRepository, Shop, Status
from laundry_protocol import reply
from money import Money
from profitina_laundry import ApiClient, Controller, SocketApi


def countdown(start):
    """Exercise 16.1: start, start-1, ..., 1, like the display of a lift going down."""
    while start >= 1:
        yield start
        start -= 1


def remove_morning(hours):
    """Exercise 16.2: a new list without the pickups before 12 o'clock."""
    return [h for h in hours if h >= 12]


def split(money, n):
    """Exercise 16.3: n parts that differ by at most one unit and add up exactly."""
    if n <= 0:
        raise ValueError(f"n must be positive: {n}")
    base, extra = divmod(money.amount, n)       # the first "extra" parts get one unit more
    return [Money(base + (1 if i < extra else 0), money.currency) for i in range(n)]


def total(request):
    """Exercise 16.6: TOTAL WASH 3 IRON 2 adds up several PRICE replies."""
    w = request.split()
    if len(w) < 3 or len(w) % 2 == 0:
        return "ERR usage: TOTAL <service> <kg> [<service> <kg> ...]"
    s = 0
    for service, kg in zip(w[1::2], w[2::2]):
        one = reply(f"PRICE {service} {kg}")
        if one.startswith("ERR"):
            return one                          # the first error decides
        s += int(one[3:])
    return f"OK {s}"
```

Listing 16.36 (Continued) *exercise_solutions.py: the same solutions in Python, and the SQLite joke store of Exercise 8.*

```
class SqliteJokeStore(JokeStore):
    """Exercise 16.8: the database member's store, on SQLite (part of Python's standard
library)."""

    def __init__(self, jokes):
        self._db = sqlite3.connect(":memory:", check_same_thread=False)
        self._db.execute("CREATE TABLE joke (n INTEGER PRIMARY KEY, clue TEXT, answer
TEXT)")
        self._db.executemany("INSERT INTO joke VALUES (?, ?, ?)",       # ? keeps data out
of SQL
                             [(i, j.clue, j.answer) for i, j in enumerate(jokes)])
        self._db.commit()

    def size(self):
        return self._db.execute("SELECT COUNT(*) FROM joke").fetchone()[0]

    def get(self, index):
        row = self._db.execute("SELECT clue, answer FROM joke WHERE n = ?",
(index,)).fetchone()
        if row is None:
            raise IndexError(f"joke {index} does not exist")
        return Joke(*row)


class TestExercises(unittest.TestCase):
    def test_countdown(self):
        self.assertEqual([4, 3, 2, 1], list(countdown(4)))
        self.assertEqual([], list(countdown(0)))

    def test_remove_morning(self):
        self.assertEqual([14, 17], remove_morning([9, 14, 11, 17, 9]))

    def test_split(self):
        parts = split(Money(10_000), 3)
        self.assertEqual(["IDR 3,334", "IDR 3,333", "IDR 3,333"], [str(p) for p in parts])
        self.assertEqual(Money(10_000), parts[0] + parts[1] + parts[2])

    def test_total(self):
        self.assertEqual("OK 31000", total("TOTAL WASH 3 IRON 2"))
        self.assertEqual("ERR unknown service SILK", total("TOTAL WASH 3 SILK 1"))

    def test_sqlite_store_keeps_the_contract(self):
        store = SqliteJokeStore([Joke("Lettuce", "Lettuce in, it's cold out here!"),
                                 Joke("Harry", "Harry up, the laundry is ready!")])
        self.assertEqual([], contract(store))
        self.assertEqual("Harry", store.get(1).clue)
```

Listing 16.36 (Continued) *exercise_solutions.py: the same solutions in Python, and the SQLite joke store of Exercise 8.*

```
    def test_ready_counter_over_the_network(self):
        shop, ready = Shop(), []
        shop.add_listener(lambda o: ready.append(o.id) if o.status is Status.READY else
None)
        api = SocketApi(Controller(shop, FileOrderRepository(os.devnull)))
        with ApiClient(api.start(0)) as c:
            c.call("PLACE Sari WASH 3")
            c.call("PLACE Budi IRON 1")
            for order_id in ("L1", "L1", "L2", "L2", "L2"):
                c.call(f"ADVANCE {order_id}")   # L1 -> READY; L2 -> READY -> PICKED_UP
        api.stop()
        self.assertEqual(["L1", "L2"], ready)


if __name__ == "__main__":
    unittest.main(verbosity=2)
```

```
test_countdown (__main__.TestExercises.test_countdown) ... ok
test_ready_counter_over_the_network
(__main__.TestExercises.test_ready_counter_over_the_network) ... ok
test_remove_morning (__main__.TestExercises.test_remove_morning) ... ok
test_split (__main__.TestExercises.test_split) ... ok
test_sqlite_store_keeps_the_contract
(__main__.TestExercises.test_sqlite_store_keeps_the_contract) ... ok
test_total (__main__.TestExercises.test_total) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.003s

OK
```

1. See `Countdown` and `countdown`. The Java iterator keeps its own `next`, starting at n; `hasNext()` is `next >= 1`. A loop over `Countdown(0)` runs zero times, because `hasNext()` is `false` at once; `list(countdown(0))` is `[]`.
2. See `removeMorningWithIterator` and `removeMorningWithRemoveIf`; both give `[14, 17]`. `Iterator.remove()` removes the element last returned by `next()` and updates the iterator's own expectations, so no exception follows; `removeIf` does the whole loop inside the list. A for-each loop that calls `list.remove` changes the list behind the iterator's back, and the next call of `next()` throws `ConcurrentModificationException`, exactly as in Problem 16.9. In Python, build a new list with a comprehension, `[h for h in hours if h >= 12]`, rather than removing while iterating.
3. With a the amount and n the number of parts, every part gets

$$b = \lfloor \frac{a}{n} \rfloor, \qquad r = a \bmod n \tag{16.7}$$

that is, every part gets b and the first r parts get one unit more, so the parts add up to $nb + r = a$. By Equation (16.7), for $a = 10{,}000$ and $n = 3$: $b = 3333$ and $r = 1$, so the parts are 3,334, 3,333 and 3,333. Multiplying by 1/3 and rounding would give three times 3,333 and lose one rupiah. 4. The largest is 6, when the threads do not interleave at all. The smallest is 2. Interleaving: thread A reads 0; thread B does two complete increments (count is 2); A writes 1; B reads 1; A does its remaining two increments (count is 3); B writes 2. Each thread's last write can undo the other's work, but each thread's own first and last increments cannot both be lost, which is why the minimum is 2 and not 1, whatever the number of increments per thread. 5. Thread 1 locks A and waits for B; thread 2 locks B and waits for A; each waits for a lock the other holds, forever. This is the dining philosophers' situation of Chapter 14 with two philosophers. The rule is a *global lock*

order [19]: give every queue a number and always lock the lower number first. Both transfers then lock A before B, and the circular wait cannot arise. (A timeout on the second lock, `tryLock` in Java, is an alternative that turns the deadlock into a retry.) 6. See `total`. `TOTAL WASH 3 IRON 2` returns `OK 31000`, because $7000 \times 3 + 5000 \times 2 = 21{,}000 + 10{,}000$. `TOTAL WASH 3 SILK 1` returns the first error, `ERR unknown service SILK`. 7. The server sends `PING` when it has sent nothing for 5 seconds; the client must answer `PONG` at once, in any state. The server sets `setSoTimeout(15_000)` on each client socket; a `SocketTimeoutException` means three missed heartbeats and the connection is closed as "client lost". In **Table 16.5**, every state gains a row (`PONG`, no reply, same state), so that a heartbeat answer never counts as a wrong line. The client, symmetrically, may treat 15 seconds without any line, heartbeat or joke, as "server lost". 8. See `SqliteJokeStore`. It creates a table `joke(n, clue, answer)`, inserts the jokes with `?` placeholders, which keep data out of the SQL text and prevent SQL injection, and answers `get(i)` with `SELECT ... WHERE n = ?`, raising `IndexError` when no row is found. The contract test of Problem 16.1 reports no failures. A Java `JdbcJokeStore` would use the JDK's `java.sql` interfaces (`Connection`, `PreparedStatement`, `ResultSet`) unchanged, but it needs a *JDBC driver* for the chosen database, a JAR file such as the SQLite or PostgreSQL driver, which the JDK does not include; that is the moment to start using Maven or Gradle (Section 16.13.2). 9. "The window says *Connected* but nothing appears": likely cause, the server accepted the connection but is not responding (overloaded, or stuck in an earlier conversation, like the one-at-a-time server of Problem 16.3); remedy, wait for the time-limit message, then restart the server and check its console. "`Too many open files`": likely cause, the server does not close the sockets of finished or departed clients, or the operating system's limit is low; remedy, a programming error to fix with `try`-with-resources or `with` around every socket (Problem 16.6), and, for a large number of genuine users, raising the operating system's limit. 10. See the end of `ExerciseSolutions.main` and `test_ready_counter_over_the_network`. The listener checks `o.status() == Status.READY`; after the five `ADVANCE` commands, L1 has passed through `READY` once and L2 has passed through `READY` on its way to `PICKED_UP`, so the count is 2. Because the listener is called from the network threads, the Java count is an `AtomicInteger`. 11. Five tests, with the stub server of Problem 16.4: (1) a whole joke, lines `Knock! Knock!`, `Lettuce`, `Lettuce in, ... (y/n)`, `Bye.` with a user answering `n`: the client wrote `Who's there?`, `Lettuce who?`, `n`; (2) the server hangs up after `Knock! Knock!`: `converse` returns without an exception and the transcript has two lines; (3) the user function returns `null`: the client stops without writing anything further; (4) lines after `Bye.` are ignored; (5) a line with non-ASCII characters (`Kopi ☕`) arrives intact through the UTF-8 reader and writer. 12. `src/main/java/LaundryCore.java` and `src/main/java/ProfitinaLaundry.java` (in a real project inside a package folder such as `src/main/java/com/profitina/laundry/`); the test class `src/test/java/ShopTest.java`, using the JUnit Jupiter API (JUnit 5, or JUnit 6, its current major version [20]) as declared in `pom.xml`. The command `mvn test` compiles both folders and runs the tests; `mvn package` also builds a JAR.

References

[1] Python Software Foundation, "unittest — Unit testing framework," *The Python Standard Library*, Python 3.14. [Online]. Available: https://docs.python.org/3/library/unittest.html

[2] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Hoboken, NJ, USA: Wiley, 2011.

[3] J. D. C. Little, "A proof for the queuing formula: L = λW," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.

[4] P. Deitel and H. Deitel, *Java How to Program, Early Objects*, 11th ed. Hoboken, NJ, USA: Pearson, 2017.

[5] J. Klensin, "Simple Mail Transfer Protocol," RFC 5321, Internet Engineering Task Force, Oct. 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5321

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

[7] E. Freeman and E. Robson, *Head First Design Patterns*, 2nd ed. Sebastopol, CA, USA: O'Reilly, 2020.

[8] The Apache Software Foundation, "Maven Getting Started Guide." [Online]. Available: https://maven.apache.org/guides/getting-started/

[9] Gradle Inc., "Gradle User Manual." [Online]. Available: https://docs.gradle.org/current/userguide/userguide.html

[10] Python Packaging Authority, "Installing packages using pip and virtual environments," *Python Packaging User Guide*. [Online]. Available: https://packaging.python.org/

[11] VMware Tanzu, "Spring Boot Reference Documentation." [Online]. Available: https://docs.spring.io/spring-boot/

[12] Stack Overflow, "2025 Stack Overflow Developer Survey: Technology — Web frameworks and technologies," Jul. 2025. [Online]. Available: https://survey.stackoverflow.co/2025/technology

[13] S. Ramírez, "FastAPI Documentation." [Online]. Available: https://fastapi.tiangolo.com/

[14] J. Bloch, *Effective Java*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2018.

[15] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Upper Saddle River, NJ, USA: Addison-Wesley, 2006.

[16] L. Ramalho, *Fluent Python*, 2nd ed. Sebastopol, CA, USA: O'Reilly, 2022.

[17] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2018.

[18] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, anniversary ed. Reading, MA, USA: Addison-Wesley, 1995.

[19] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, no. 2, pp. 115–138, 1971.

[20] The JUnit Team, *JUnit User Guide*, version 6.1.3 (Aug. 2026), and "Release notes: 6.0.0" (Java 17 baseline). [Online]. Available: https://docs.junit.org/current/

[21] Oracle, "The Java Tutorials: All About Sockets" and "Lesson: Concurrency." [Online]. Available: https://docs.oracle.com/javase/tutorial/

[22] Python Software Foundation, "socket," "threading," "queue" and "dataclasses," *The Python Standard Library*, Python 3.14. [Online]. Available: https://docs.python.org/3/library/

[23] Oracle, "Lesson: Concurrency in Swing," *The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/uiswing/concurrency/

The socket classes, the knock-knock protocol and its multi-client server follow the Java tutorial [21], and the Python versions use the standard library's socket, threading, queue and dataclass modules [22]; the rule that Swing components are used only on the event dispatch thread is from the Swing concurrency lesson [23]. Races, monitors, `wait`/`notifyAll`, atomic variables and the design of thread-safe classes follow Goetz et al. [15] and Bloch [14], who also gives the hash contract and the case for immutable value objects; the producer–consumer example follows Deitel and Deitel [4], and the global lock order that prevents deadlock goes back to Dijkstra [19]. Little's law is from Little [3]; the dot-terminated reply is the convention of SMTP [5]; mutation testing and test design are treated by Myers et al. [2]; iterators and generators in Python are explained in depth by Ramalho [16]. The design patterns are those of Gamma et al. [6] and Freeman and Robson [7]; refactoring is from Fowler [17], and the lessons on teams from Brooks

[18]. Maven [8], Gradle [9], pip and virtual environments [10], Spring Boot [11] and FastAPI [13] are documented by their projects; the Python tests use `unittest` [1].