Object-Oriented Programming (EF234302)

Lecture 1 — Objects, Programs, and Your First Workbench

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*, Chapter 1

Lecturer: Irfan Subakti — `yifana@gmail.com`

Department of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya

Today a laundry shop on a street corner in Surabaya becomes a society of objects, and the first programs run in both languages. Book: Chapter 1.

Agenda

1. What a program is: a recipe, two formulas, and how source code becomes a running program.

2. Thinking in objects: state, behavior, identity; classes as blank forms; messages; abstraction.

3. Why procedures are not enough: the notebook program and its two careless lines.

4. Why Java and why Python; the JDK, the interpreter, and VS Code on Windows, macOS and Linux.

5. First programs, a first debugging session, and a first class: the laundry `Order` that refuses nonsense.

Book: Chapter 1, Sections 1.1–1.14.

Where We Begin

Bu Sari's shop: a counter with a scale, three washing machines, two dryers, a shelf of bags, and a thick notebook.

Already full of our ideas: a **procedure** (weigh, ask, write, price, wash, dry, fold, hand over), **information** (name, weight, service, price, status), and **things** that each know something and can do something.

When Bu Sari hires a second worker she explains the shop *one kind of thing at a time*: how a ticket works, how a machine works. That is object-oriented programming.

> **KEY INSIGHT.** A program can be one long list of instructions that shuffles numbers, or a small society of **objects** that each look after their own information. The second way is how most software you use every day is organized.

Every idea is shown in words, then in Java, then in Python, then run, then explained line by line. Book: Section 1.1.

A Recipe for the Counter

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
```

Each line says one thing; “round” and “floor” leave nothing to taste; line 5 runs only *if* line 4 is true. The changing values get short names, w and p.

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.

Pseudocode 1.1 is the card Bu Sari tapes to the counter for her nephew Dimas. Book: Section 1.2.1 and Pseudocode 1.1.

The Price as Two Formulas

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

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

$\lfloor x \rfloor$, "floor of x", is the largest whole number not bigger than x: $\lfloor 12250.0 \rfloor = 12250$, $\lfloor 7.9 \rfloor = 7$.

BY HAND. Ani, 3.5 kg regular, by Equation (1.1): $p(3.5) = \text{round}(24500) = 24500$. Budi, 5 kg express, by Equation (1.2): $p(5) = 35000$, then $35000 + \lfloor 35000/2 \rfloor = 52500$.

Remember Rp 24,500 and Rp 52,500: the computer prints them several times today. Book: Section 1.2.1, Equations (1.1) and (1.2).

Languages for People and for Machines

The processor understands only **machine language**: long sequences of numbers, each meaning something tiny.

A **programming language** is precise enough to translate without guessing and readable enough to explain to a colleague. Line 3 of the card, in both:

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

Java says what *kind* of value `p` holds (`long`) and ends statements with `;`. Python lets the value decide its kind and ends at the end of the line.

The text you write is **source code**; `.java` and `.py` source files are plain text.

Chapter 3 explores these differences carefully. Book: Section 1.2.2.

From Source Code to a Running Program

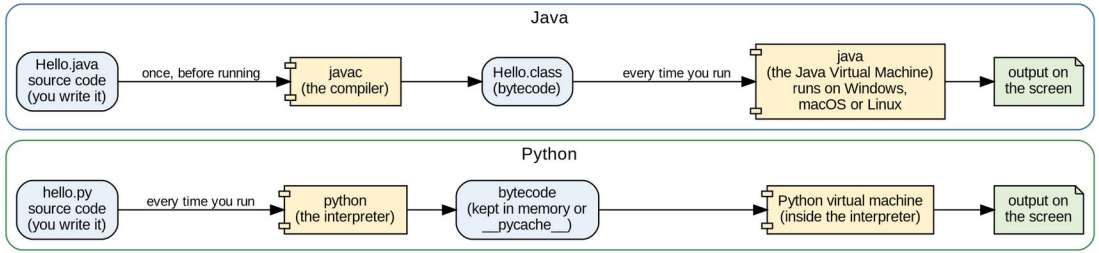


Figure 1.1 How source code becomes a running program: Java compiles once to bytecode that the Java Virtual Machine runs; Python compiles to bytecode each time it runs, inside the interpreter.

Figure 1.1: a compiler translates the whole book once; an interpreter translates aloud as you go. Java and Python each use a little of both. Book: Section 1.2.3 and Figure 1.1.

Compile, Then Run — or Just Run

Java: `javac Hello.java` writes `Hello.class`, which holds **bytecode** for the **Java Virtual Machine**; `java Hello` starts a JVM that runs it. Same bytecode on every machine: *write once, run anywhere*.

Python: `python hello.py` compiles to Python bytecode inside the interpreter (imported modules are cached in `__pycache__`) and runs it at once.

Data and operations. Every program is data (3.5 kg, "Ani", express or not) plus operations on it. Procedural code keeps them apart; object-oriented code keeps each piece of data *with* the operations that belong to it.

WARNING. In Java, after changing the source you must compile again, or you will run the old version.

Book: Sections 1.2.3–1.2.4.

Looking Around the Shop

Ask of every *thing* two questions: **what does it know?** and **what can it do?**

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

Table 1.1, and not any code, is where an object-oriented program begins. Book: Section 1.3.1 and Table 1.1.

State, Behavior, and 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).

Two customers with bags of the same weight, same service, same price: still **two tickets**. Handing Ani's clothes to Budi is a serious mistake.

State changes over time: RECEIVED at seven, WASHING at ten, READY at two. The ticket is the same ticket throughout.

Behavior is *how* state changes, through well-defined steps — and some steps are **refused** at some moments: you cannot pick up laundry still in the machine.

Book: Section 1.3.2.

Classes: the Blank Form

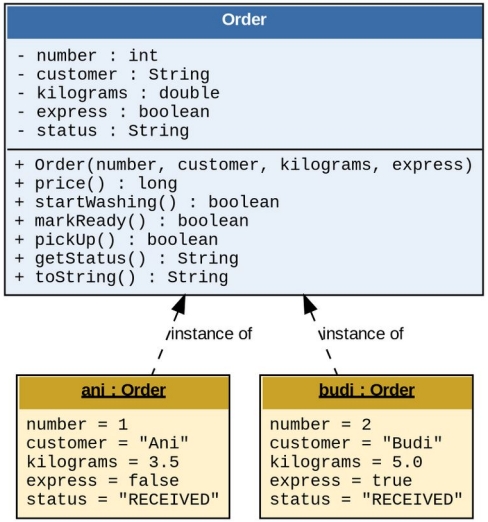


Figure 1.2 The class `Order` (top) is the blank form: it lists the state every order holds and the behavior every order offers; the objects `ani` and `budi` (bottom) are two filled-in tickets.

Figure 1.2 in UML: the class is the blank form printed on a pad; `ani` and `budi` are two filled-in tickets. Book: Section 1.3.3 and Figure 1.2.

Reading the Class Box

Top: the class name. **Middle:** the **fields** with their types — `int`, `String`, `double`, `boolean`; the minus sign means *private*.

Bottom: the **methods**, with a plus sign for *public*. `Order(number, customer, kilograms, express)` is the **constructor**, which fills in a new form.

Object boxes: underlined `name : Class`, showing *values*. Each object has its own copy of every field; changing Ani's status leaves Budi's alone.

> **KEY INSIGHT.** A class is written once, as source code. Objects are created while the program runs, as many as needed. "Write a class" means design the blank form; "create an object" means fill one in.

*An object is also called an **instance** of its class. Book: Section 1.3.3.*

Messages: Objects Working Together

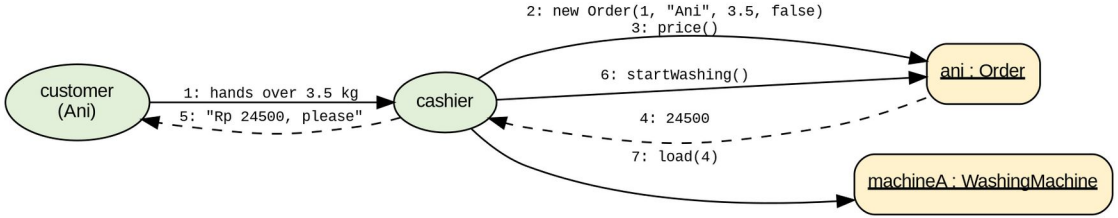


Figure 1.3 One morning at the counter as a conversation between objects: solid arrows are messages, dashed arrows are answers, and the numbers give the order in which they happen.

In Figure 1.3 solid arrows are messages, dashed arrows answers, and the numbers give the order. Book: Section 1.3.4 and Figure 1.3.

Following the Numbers

1 the customer hands over 3.5 kg; **2** the cashier creates `new Order(1, "Ani", 3.5, false)`; **3** asks the order for its price; **4** the order answers `24500`, computed by Equation (1.1) from its *own* weight.

5 the cashier tells the customer; **6** tells the order washing has started; **7** loads four kilograms into machine A.

The cashier **does not compute the price**: it asks the order. If the price list changes next year, only the order's `price()` changes.

Message names are method names, written with a dot: `ani.price()` sends *price* to `ani`. The word comes from Smalltalk.

Java and Python programmers usually say "call a method on an object"; it means the same thing. Book: Section 1.3.4.

Abstraction, and the Four Big Ideas

Abstraction: keep only what matters for the job — no coffee stain, no crease. Too little and the model cannot work; too much and it is costly and confusing. The *what* is public; the *how* is the object's own business.

Encapsulation: state bundled with, and hidden behind, the behavior that uses it (Chapters 3 and 7).

Interfaces make abstraction sharp (Chapter 6). **Inheritance:** an `ExpressOrder` is an `Order` (Chapter 7).

Polymorphism: the same message, `price()`, answered correctly in its own way by each kind of order (Chapter 7).

Everything in the rest of the book grows from these few ideas. Book: Sections 1.3.5–1.3.6.

The Notebook Program, in Java

Listing 1.1 `LaundryLedger.java`, part 1: four parallel arrays in `main`, and two careless lines.

```
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);
    }
```

*Listing 1.1 is **procedural programming**: an **array** is a numbered row of boxes, and position `i` of every array describes order `i`. Book: Section 1.4.1 and Listing 1.1.*

The Two Stand-Alone Functions

Listing 1.2 `LaundryLedger.java`, part 2: the price function and the printing function, which receive the arrays from outside.

```
    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]);
        }
    }
}
```

The data lies in the open in `main`; the operations live elsewhere and receive four arrays each time.

Listing 1.2 completes the class; `base / 2` between two whole numbers is already the floor of Equation (1.2). Book: Section 1.4.1 and Listing 1.1.

The Same Notebook in Python

Listing 1.3 `laundry_ledger.py`, part 1: the same two functions over four parallel Python lists.

```
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])
```

```
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
```

The `main` function of Listing 1.3 builds the four lists exactly as the Java version does; both programs print the same seven lines. Book: Section 1.4.1 and Listing 1.2.

What Went Wrong

The first three lines are right: $p(3.5) = 24500$, $p_{\text{express}}(5) = 52500$, $p(2) = 14000$.

Then Budi weighs **minus five kilograms** and costs **minus Rp 52,500**; Citra is **PICKED UP** although her clothes were never washed.

Grown to twenty thousand lines by five people, *any* line may write into `status` or `kilograms`. The rule "status only moves forward" lives only in programmers' heads.

The four arrays must stay *in step*; adding a telephone number means a fifth array and changing every function.

> **WARNING.** "Just be careful" is not a strategy. It fails, reliably, for any program that is shared, changed or kept for more than a few weeks.

Book: Section 1.4.2.

The Object Answer

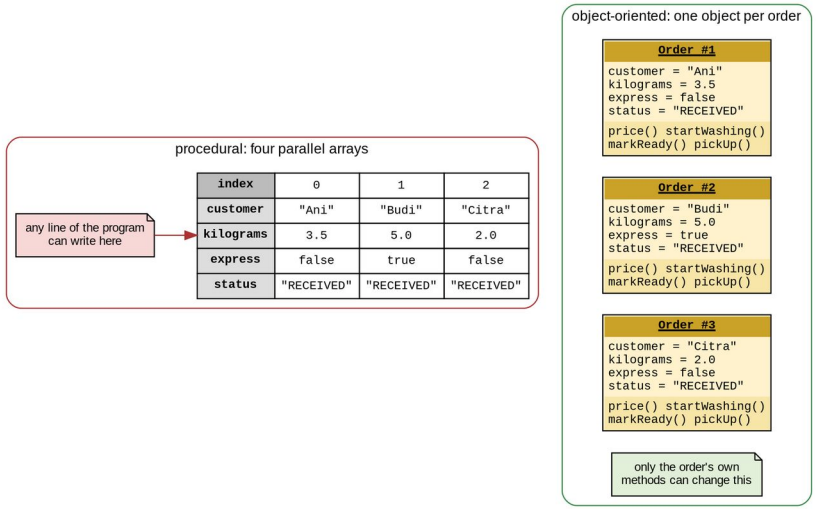


Figure 1.4 The same three orders organized two ways: on the left, four parallel arrays that any line of the program can write to; on the right, one object per order, whose state only its own methods can change.

On the right of Figure 1.4 each order is a closed box with four buttons, and `pickUp()` looks at the status first and refuses if the clothes are not ready. Object orientation is not new arithmetic; it is a way to organize the same arithmetic. Book: Section 1.4.3 and Figure 1.4.

Why Java

Created at Sun Microsystems by James Gosling's team; released in 1995; stewarded by Oracle since 2010, with the open **OpenJDK**.

Grammar borrowed from C and C++, minus pointer arithmetic and manual freeing of memory; the **garbage collector** reclaims unused objects.

Portable (WORA), **supported everywhere**, **widely used** for client–server systems, and **explicit** about types and access — the grammar spells out the ideas of objects.

LTS releases 8, 11, 17, 21 and 25. This book targets **Java 25**; every program also runs on 17 and 21.

Book: Sections 1.5.1–1.5.2.

And Why Python Too

Guido van Rossum, 1991: code is read more often than it is written. No braces — **indentation** shows the block.

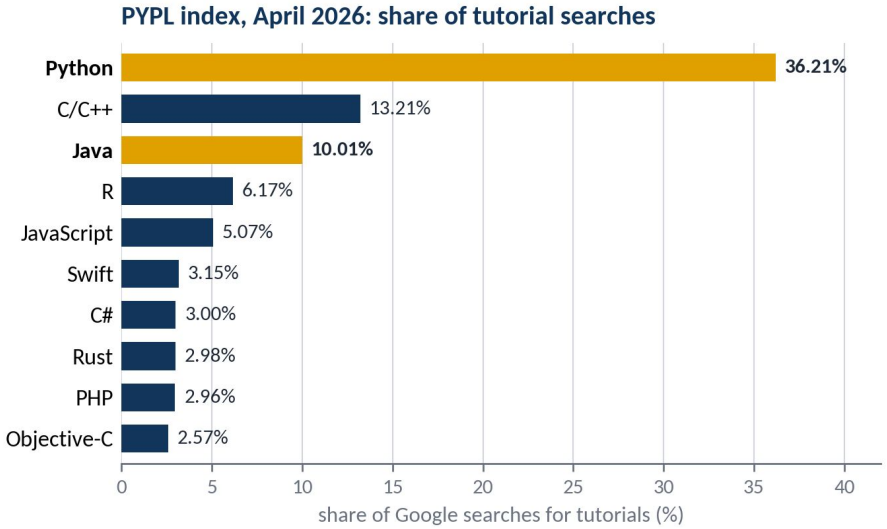
Dynamic typing (checked while running) against Java's **static typing** (checked when compiling). In Python every value, even 7, is an object.

The language of data analysis, scientific computing, automation, AI and machine learning. This book targets **Python 3.14**; every program also runs on 3.10+.

Java shows objects with every joint visible; Python shows the same ideas with the scaffolding removed. Seeing both, you learn the idea, not one spelling.

Book: Section 1.5.3.

What the Rankings Say: PYPL

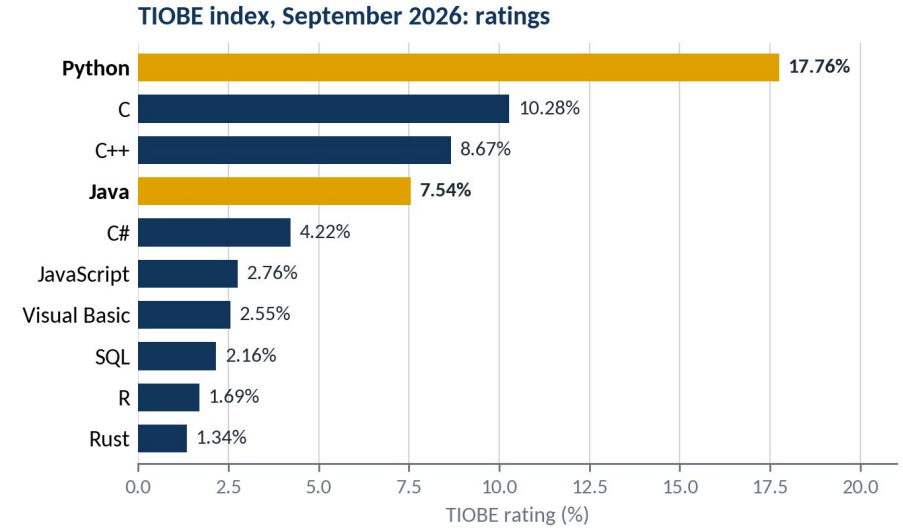


Source: PYPL PopularitY of Programming Language index, worldwide, April 2026.

Figure 1.5 The PYPL ranking for April 2026, based on how often each language's tutorials are searched for: Python first, Java third.

Figure 1.5 counts tutorial searches: Python 36.21 % *, C/C++* 13.21 % *, Java* 10.01 % *(April 2026). The Stack Overflow Developer Survey 2025 puts Python fourth (*57.9 % *of respondents) and Java seventh (*29.4 % *). Book: Section 1.5.4 and Figures 1.5–1.6.*

What the Rankings Say: TIOBE



Source: TIOBE Index for September 2026.

Figure 1.6 The TIOBE ranking for September 2026, based on engineers, courses and vendors per language: Python first, Java fourth.

Figure 1.6 counts engineers, courses and vendors: Python 17.76 %, C 10.28 %, C++ 8.67 %, Java 7.54 % (September 2026). GitHub's Octoverse 2025 counts monthly contributors: TypeScript, Python, JavaScript, then Java. The measurements disagree on first place, but Python is at or near the top of every one and Java in the top seven. Book: Section 1.5.4 and Figures 1.7–1.8.

The Two Languages Compared

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

The most important row of Table 1.2 is missing: the ideas are the same. Learn an idea first, then its Java spelling, then its Python spelling. Book: Sections 1.5.5–1.5.6 and Table 1.2.

The Tools of the Trade

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

Since Java 11 the **JDK** contains its own runtime: install the JDK and nothing else. This book recommends **Eclipse Temurin**. Python's distribution brings the interpreter, the standard library, IDLE and **pip**.

*Table 1.3 lists the JDK tools the book uses; a Python **virtual environment** (`.venv`) is optional here. Book: Sections 1.6.1 and 1.6.3 and Table 1.3.*

The `bin` Folder and the PATH



Figure 1.7 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`.

*In Figure 1.7, `javac` compiles and `java` runs (`javac.exe` and `java.exe` on Windows). A terminal finds them only if `bin` is on the **PATH**; `set path=D:\Program\Java\JDK\bin;%path%` adds it for one window. "Not recognized" means: not on the PATH of this window. Book: Section 1.6.2 and Figures 1.9–1.11.*

The Workbench on Three Systems

1. **JDK:** Temurin 25 from adoptium.net — `.msi` (or `winget install EclipseAdoptium.Temurin.25.JDK`), `.pkg` (or `brew install --cask temurin@25`), or `sudo apt install temurin-25-jdk`.
2. **Python 3.14:** python.org installer (tick **Add python.exe to PATH**), `brew install python@3.14`, or `sudo apt install python3.14 python3.14-venv`.

3. Check in a new terminal, **4. install VS Code**, **5. add the two extensions**.

6. Run: `javac Hello.java` then `java Hello`; `python hello.py` (`python3` on macOS and Linux).

Install the language tools first, check them in a terminal, then install the editor and connect the two; take the newest update of the version named. Fedora and Arch commands are in the book. Book: Section 1.7 and Table 1.5.

Check Both Tools in a New Terminal

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

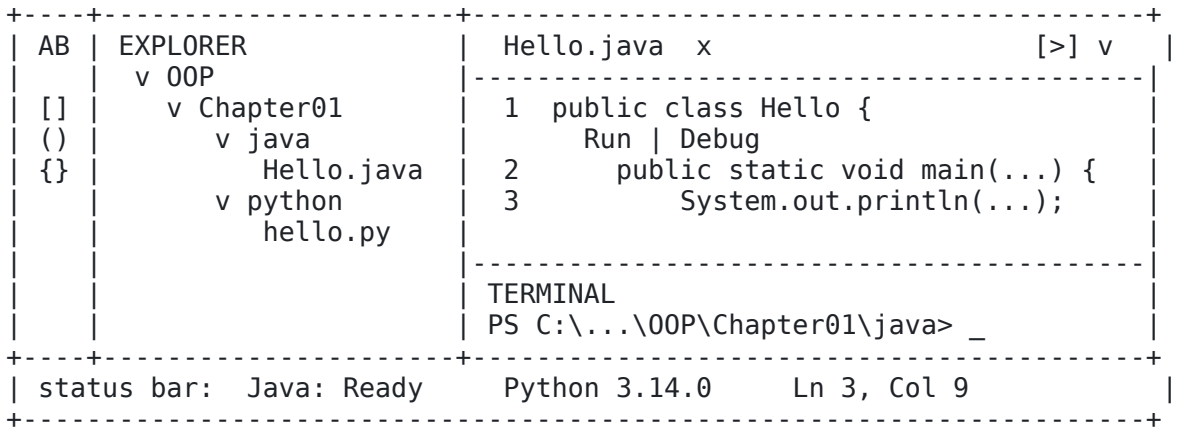
Close every old terminal first: old windows keep the old PATH. On macOS and Linux type `python3` (or `python3.14`).

Windows: tick **Add python.exe to PATH**; if `python` opens the Microsoft Store, use `py` or switch off the *app execution aliases*.

macOS: Temurin `.pkg` (aarch64 or x64), Python *universal2* installer, then *Install Certificates.command*. **Linux:** Adoptium repository and `temurin-25-jdk`; Python 3.14 *beside* the system Python.

Book: Sections 1.7.1–1.7.3.

VS Code and Its Two Extensions



Install **Extension Pack for Java** and **Python** (both by Microsoft); check **Java: Configure Java Runtime** and **Python: Select Interpreter**.

Activity bar, side bar, editor with its Run button, panel with the Terminal, status bar. One folder per chapter: `OOP/Chapter01/java`, `OOP/Chapter01/python`. *Book: Section 1.7.4.*

Hello, in Both Languages

Listing 1.4 `Hello.java`: the first Java program, which greets the reader and reports which Java version is running it.

```java
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") + ".");
    }
}
```

Listing 1.5 `hello.py`: the first Python program, which does the same as Listing 1.4.

```python
import platform

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

Run Listing 1.4 with the Run button, with `javac Hello.java` then `java Hello`, or with `java Hello.java`; run Listing 1.5 with `python hello.py`. The file name must match the class, capital H included. Book: Sections 1.7.5–1.7.6 and Listings 1.3–1.4.

Reading Hello Line by Line

`public class Hello {` — all Java code lives in a class. `public static void main(String[] args)` — the launcher's entry point: *public* so it may be called, *static* so no object is needed, *void* because it returns nothing.

`System.out.println(...)` — `out` is an object inside the class `System`; `println` is a message sent to it with a dot. `+` joins strings.

Python: `import platform` brings in a **module**; `print` needs no class, no `main`, no semicolon; the file runs from the top.

> **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.

The second output line is honest about which Java or Python actually ran. Book: Sections 1.7.5–1.7.6.

How Many Machine Loads?

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

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

A machine holds $c = 8$ kg. $\lceil x \rceil$ is the smallest whole number not less than x: 9 kg needs $\lceil 1.125 \rceil = 2$ loads. Whole-number division gives a floor, so Equation (1.4) turns it into a ceiling.

Equation (1.3) is what we want; Equation (1.4) is how a computer gets it from integer division. Book: Section 1.7.7 and Equations (1.3)–(1.4).

Why the Ceiling Trick Works

Write $w = q\,c + r$ with $0 \le r \le c - 1$.

If $r = 0$: $(w + c - 1)/c = q + (c - 1)/c$, and $(c - 1)/c < 1$, so the floor is $q = w/c$.

If $r \ge 1$: the ceiling is $q + 1$, and $(w + c - 1)/c = q + (r + c - 1)/c$ with $c \le r + c - 1 \le 2c - 2 < 2c$, so the floor is $q + 1$.

CHECK. $c = 8$: $w = 9$ gives $\lfloor 16/8 \rfloor = 2$; $w = 8$ gives $\lfloor 15/8 \rfloor = 1$; $w = 3$ gives $\lfloor 10/8 \rfloor = 1$. Both cases agree with Equation (1.3).

Book: Section 1.7.7.

Counting Loads in Java

Listing 1.6 `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);
    }
}
```

By hand: $1+2+1+2+1=7$*, exactly what Listing 1.6 prints on the next page. Book: Section 1.7.7 and Listing 1.5.*

Counting Loads in Python

Listing 1.7 `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))
```

```
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
```

Both listings print these six lines. In Listing 1.7 `//` is whole-number division and `str(...)` turns a number into text; there is no class and no `main`. Book: Section 1.7.7 and Listing 1.6.

Watching the Program Think

1. Breakpoint: click left of `totalLoads = totalLoads + loads;` — a red dot. **2. Start:** *Debug* above `main`, or `F5`; the yellow line has *not yet* run.

3. Variables panel: `capacity = 8`, `totalLoads = 0`, `i = 0`, `loads = 1`.

4. Step: `F10` runs one line; `F5` continues to the next pass. Watch the total grow: 1, 3, 4, 6, 7.

5. Watch `orders[i] / capacity`: for `i = 1`, Java shows `1`, Python shows `1.125` — which is why the two listings divide differently. **6. Stop:** `Shift+F5`.

`F11` steps into a method call and `Shift+F11` steps out; Chapter 2 uses them all. Book: Section 1.7.7.

When Something Goes Wrong

`'java' is not recognized` / `command not found` — `bin` not on this terminal's PATH: open a *new* terminal, or reinstall with **Add to PATH**.

`class Hello is public, should be declared in a file named Hello.java` — rename the file to match the class exactly.

`Could not find or load main class Hello.class` — type `java Hello`, naming the class, not the file.

`IndentationError: unexpected indent` — line the block up with spaces only. VS Code runs the wrong Python — **Python: Select Interpreter**.

Read the first line of the message, find it in Table 1.6, apply the fix before trying anything else. Book: Section 1.7.8 and Table 1.6.

A Receipt: Variables and Arithmetic

Listing 1.8 `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);
    }
}
```

*In Listing 1.8 every variable is **declared** with its type; `long` holds up to about 9.2×10^{18}, so money never overflows. $50000 - 24500 = 25500$. Book: Section 1.8.1 and Listing 1.7.*

The Same Receipt in Python

Listing 1.9 `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))
```

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

*Listing 1.9 has no type words, whole numbers without limit, and an explicit `str(...)`. Java names are **camelCase** (`pricePerKg`), Python names **snake_case** (`price_per_kg`). Book: Section 1.8.1 and Listing 1.8.*

Designing the `Order` Class

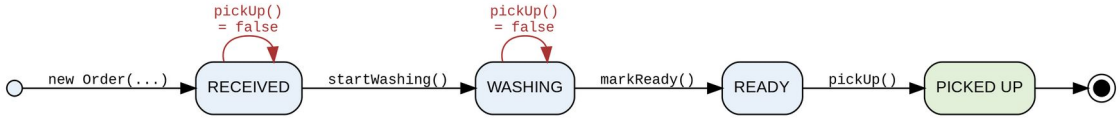


Figure 1.8 The life of an order: each message moves it one step forward, and a message sent at the wrong moment, such as `pickUp()` while the laundry is still washing, is refused and changes nothing.

Knows: number, customer, kilograms, express, status. Does: price, start washing, mark ready, pick up. Must never: skip a step or go back. The red loops in Figure 1.8 say that a message at the wrong moment returns `false` and changes nothing. Book: Section 1.8.2 and Figure 1.12.

`Order.java`: Fields and Constructor

Listing 1.10 `Order.java`, part 1: the five private fields and the constructor.

```
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";
    }
```

`private`: no other class may touch a field — `ani.status = "PICKED UP";` elsewhere does not compile. `final`: set once, in the constructor. `this.number` is the field, plain `number` the parameter.

Listing 1.10: the constructor has no return type, runs once per new object, and always starts the status at RECEIVED. Book: Section 1.8.3 and Listing 1.9.

`Order.java`: Price and the First Step

Listing 1.11 `Order.java`, part 2: the price by the two formulas, and a status method that checks before it acts.

```
    // 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;
    }
```

In Listing 1.11, `price()` needs no parameters — everything is inside the object — and `base / 2` is the floor of Equation (1.2). The pattern is check, then act. Book: Section 1.8.3 and Listing 1.9.

`Order.java`: Ready and Picked Up

Listing 1.12 `Order.java`, part 3: the last two status methods, each checking the status first.

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

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

`!status.equals("READY")` asks “is the status *not* READY?” If so, `return false;` ends the method at once and the status is untouched.

In Listing 1.12 strings are compared with `equals`, not with `==`; the Common Mistakes page explains why. Book: Section 1.8.3 and Listing 1.9.

`Order.java`: a Getter and `toString`

Listing 1.13 `Order.java`, part 4: a getter for the status, and the one-line description.

```
    public String getStatus() {
        return status;
    }

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

`getStatus()` is a **getter**: outsiders may *look* at the status, but no method lets them *set* it to an arbitrary value.

`toString()` is called automatically when an order is printed or joined to a string; `@Override` says we deliberately replace the dull default `Order@1b6d3586`. `express ? " express" : ""` is the **conditional operator**.

Listing 1.13 closes the class. It has no `main`: it is a blank form, not a program. Book: Section 1.8.3 and Listing 1.9.

`order.py`: Constructor and Price

Listing 1.14 `order.py`, part 1: `__init__` creates the fields, and `price` computes by the same formulas.

```
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
```

`class Order:` with a colon and indentation; the constructor is `__init__`; `self` is Java's `this`, written as the first parameter of every method.

In Listing 1.14 fields come into existence when first assigned; `base // 2` is the floor of Equation (1.2) directly. Book: Section 1.8.4 and Listing 1.10.

`order.py`: the Status Methods

Listing 1.15 `order.py`, part 2: the three status methods.

```
    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
```

*In Listing 1.15 privacy is a **convention**: the leading underscore of `_status` means "do not touch from outside", and tools such as Pylance flag a violation. `!=` compares string contents safely. Book: Section 1.8.4 and Listing 1.10.*

`order.py`: a Getter and `__str__`

Listing 1.16 `order.py`, part 3: the getter and `__str__`, Python's counterpart of `toString`.

```
    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 conditional expression is written in words: `" express" if self._express else ""`. Methods are named in snake_case: `start_washing`, `mark_ready`, `pick_up`, `get_status`.

With Listing 1.16 the Python class is complete: the ideas are identical to Java's, the spelling differs in five places. Book: Section 1.8.4 and Listing 1.10.

Bringing Orders to Life

Listing 1.17 `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);
    }
}
```

Compile both files, `javac Order.java OrderDemo.java`, then `java OrderDemo`. The premature pick-up in Listing 1.17 returns `false`. Book: Section 1.8.5 and Listing 1.11.

The Same Program in Python

Listing 1.18 `order_demo.py`: the same program in Python, importing the class from `order.py`.

```python
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)
```

Same seven lines of output, except `False` and `True` with capitals. `order.py` and `order_demo.py` must share a folder.

Listing 1.18 imports the class with `from order import Order`; there is no `new`. Book: Section 1.8.5 and Listing 1.12.

Tracing the Demo with the State Diagram

```
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
```

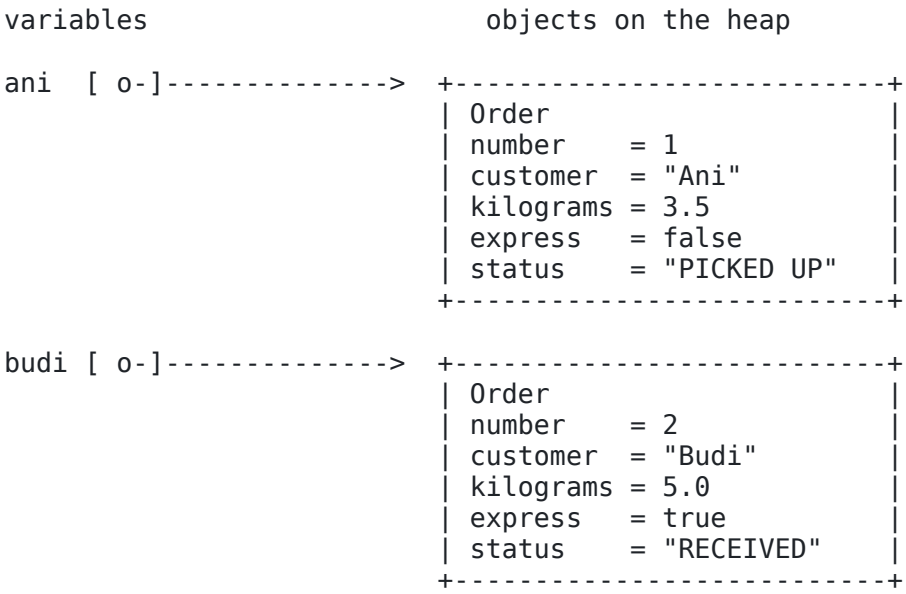
`new Order(...)` creates the object and runs the constructor; printing calls `toString()`. `ani.pickUp()` while RECEIVED is the red loop of Figure 1.8: `false`, status unchanged.

Two black arrows, then `pickUp()` returns `true`. Budi is still RECEIVED: every message went to `ani` — **identity** at work.

> **EXAMPLE.** Add `ani.status = "PICKED UP";` to `OrderDemo.java`. The compiler refuses: `error: status has private access in Order`. The silent mistake of Listing 1.1 is now impossible.

The Java output above; Python prints `False` and `True`. Book: Section 1.8.5.

What a Variable of a Class Type Holds



`ani` holds a **reference** to an object on the **heap**; `Order x = ani;` copies the reference, not the order. Python behaves the same way.

References are a safer cousin of C pointers: no arithmetic on them. Book: Section 1.8.6.

What the Class Has Bought Us

The ledger had the same data and formula, but its rules lived nowhere. The class keeps its one rule in exactly one place, and every program that uses orders gets it for free.

A new step such as IRONING is added in one class; every program follows.

Not finished: the constructor still accepts -5 kg (Chapters 3 and 5), and a slip such as `"REDAY"` inside the class would go unnoticed (later chapters replace the string).

Good classes grow one careful step at a time. Book: Section 1.8.7.

Profitina in Practice

Profitina (`app.profitina.com`) is an **AI visibility (GEO)** platform, written in Python with FastAPI.

Signals are objects: each thing the scanner measures is its own class; the scanner is the cashier of Figure 1.3, sending every signal the same messages. A new signal is one new class.

The AI model sits behind an object (Qwen2.5, locally, through Ollama): when no model is available it *says so*, like `pickUp()` refusing.

The law is part of the design: under UU PDP 27/2022, one class guarding personal data means one place for consent, export and deletion. The app advises; the user decides.

Book: Section 1.10.

A Day at Profitina Laundry

Listing 1.19 `ShopDay.java`, the day itself: four orders, moved through the day by messages, with an early pick-up refused.

```
        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();
```

Listing 1.19 uses the unchanged `Order` *class;* `for (Order o : orders)` *visits each order in turn. Book: Section 1.10.1 and Listing 1.13.*

The Day's Cash

Listing 1.20 `shop_day.py`, the evening count: cash from the collected orders, and the orders still waiting.

```
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))
```

```
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
```

Listing 1.20 is the Python spelling of the same count. Dewi: $52500+26250=78750$ *by Equation (1.2); cash* $24500+14000=38500$*; two orders still in the shop.* `ShopDay` *needed no line of its own to refuse Dewi — the class did it. Book: Section 1.10.1 and Listings 1.13–1.14.*

Common Mistakes

Running the old version — compile again after every edit, or use Run / `java Hello.java`. **File and class names that differ**, even in capitals.

Case and punctuation: `system.out`, `Print("hi")`, a missing `;`, stray indentation. Read the named line *and the one before it*.

`==` on strings in Java compares references; use `equals`. **Public fields "to save time"** invite the ledger's disaster.

Class versus object: the class has no status; each object has its own. **Regional number formats:** `String.format("%.1f", 3.5)` may print `3,5`.

Believing a program because it runs: the ledger ran perfectly and printed a negative price. Check outputs by hand against the formulas.

Book: Section 1.11.

Summary — Five Things to Remember

1. A program is an algorithm in a programming language. Java compiles to bytecode for the JVM (`javac`, then `java`); Python compiles and runs inside the interpreter.

2. An object has state, behavior and identity; a class is the blank form; objects cooperate by messages written with a dot.

3. Parallel arrays put no one in charge; an object bundles its data with the only methods allowed to change it, and those methods can **refuse**.

4. The workbench: JDK 25 (Temurin), Python 3.14, VS Code with two extensions; breakpoint, `F5`, `F10`, `F11`, Variables.

5. `Order`: private fields, a constructor, `price()` by Equation (1.1) and Equation (1.2), and status methods that follow Figure 1.8. A variable holds a reference.

Book: Section 1.12.

Next Time

Chapter 2 — The Development Environment: Editing, Running, and Debugging.

What really happens between *Run* and the first line of output: `javac`, bytecode, `javap`, the JVM; Python's hidden compiler and `dis`.

Eclipse and VS Code side by side; projects, packages and imports; and debugging from zero on a laundry bill that is Rp 16,200 too high.

Before we meet: Exercises 1.3, 1.4, 1.7 and 1.9 — prices by hand, loads by hand, a swapped pair of messages, and your own `WashingMachine`. Solutions are in Section 1.14.

Book: Sections 1.13–1.14, then Chapter 2.

Object-Oriented Programming (EF234302)

Lecture 2 — The Development Environment: Editing, Running, and Debugging

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*, Chapter 2

Lecturer: Irfan Subakti — `yifana@gmail.com`

Department of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya

Today we open the workbench itself, and then use it to find a real bug step by step instead of guessing. Book: Chapter 2.

Agenda

1. What an IDE is, and what really happens when you press *Run*: `javac`, bytecode, the JVM; Python's hidden compiler.

2. Eclipse and VS Code side by side: workspace, views, editors, perspectives, projects; folders, interpreters, `launch.json`.

3. Projects, packages, modules and imports — and the missing import in both languages.

4. Debugging from zero: breakpoints, stepping, the call stack, variables, watches, conditional breakpoints.

5. Three worked hunts: a bill Rp 16,200 too high, a day Rp 6,000 short, and a program that crashes.

Book: Chapter 2, Sections 2.1–2.18.

Where We Left Off

Chapter 1 put a workbench on your desk — JDK 25, Python 3.14, VS Code — ran first programs, watched a loop in the debugger, and built the `Order` class.

Now we slow down and look at the workbench itself, because you will spend hundreds of hours at it.

Three questions: **What happens when I run a program? How do I drive my tools well? When the program is wrong, how do I find out why?**

> **KEY INSIGHT.** You need not remember every menu path. Keep the *vocabulary* — workspace, project, package, perspective, breakpoint, step over, step into, call stack — and the *habit* of reaching for the debugger instead of guessing.

Book: Section 2.1.

What an IDE Is

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.

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

Integrated is the word that matters: the editor knows the compiler, the debugger knows both. Table 2.1 names the five jobs; every menu belongs to one of them. Book: Sections 2.2.1–2.2.2 and Table 2.1.

Two Workbenches, One Set of Ideas

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+.)

Keep Table 2.2 at hand. Book: Section 2.2.3 and Table 2.2.

Compilers, Interpreters, and When Each Complains

The CPU understands only **machine code**, and x86-64 code does not run on ARM64. Something must translate `total = total + line`.

> **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.

The book translator finds a nonsensical sentence before anyone reads the book; the live translator finds it only when the reader reaches it.

*Both Java and Python compile source into **bytecode** first, then a **virtual machine** interprets it. Book: Sections 2.3.1–2.3.2.*

From Source Code to the Processor

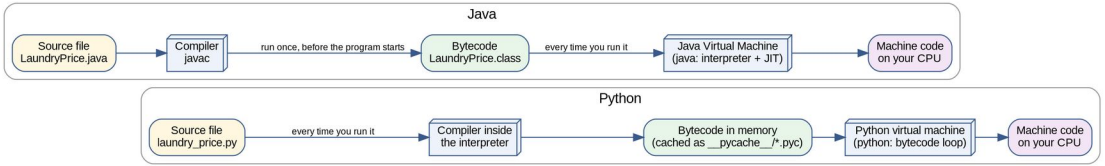


Figure 2.1 From source code to the processor. Java compiles once with javac and the JVM runs the bytecode every time; Python compiles inside the interpreter every time the program starts, then its virtual machine runs the bytecode.

Figure 2.1: Java compiles once and the JVM, with its just-in-time compiler for "hot" methods, runs the bytecode every time; Python compiles inside the interpreter each time the program starts. Book: Section 2.3.2 and Figure 2.1.

Java: `javac`, Then `java`

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

```java
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));
    }
}
```

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

```
2 + 3 = 5
```

The first command writes `Add.class` and prints nothing if all is well. Since Java 11, `java Add.java` compiles Listing 2.1 in memory and runs it, leaving no `.class` file. Book: Section 2.3.3 and Listing 2.1.

Reading the Bytecode with `javap -c`

```
  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
```

`iload_0`, `iload_1` push `a` and `b` on a small stack; `iadd` pops both and pushes the sum; `ireturn` returns it. That is `return a + b;` for a stack machine.

The JVM runs the `.class` file, not your `.java` file: edit without recompiling and you run the old program. The full listing also shows a default constructor `Add()` you never wrote. Book: Section 2.3.3.

Python: the Hidden Compiler

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)
```

```
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
```

Listing 2.2 on Python 3.11: `LOAD_FAST`, `LOAD_FAST`, `BINARY_OP (+)`, `RETURN_VALUE`. The JVM has one add per type (`iadd`, `dadd`); Python's single `BINARY_OP` decides at run time what "plus" means. Imported modules are cached in `__pycache__`. Book: Section 2.3.4 and Listing 2.2.

When Mistakes Are Caught

A missing `;` in Java and a missing `:` in Python are both caught **before** anything runs — file, line, caret, kind:

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

A name that does not exist is different: Java's compiler checks every name and type; Python looks names up only when the line runs.

> **KEY INSIGHT.** *Java checks names and types before running; Python checks them while running.* Knowing which one you are in tells you where to look for a mistake.

A misspelled name in a rarely used Python branch can hide for months — one reason testing (Chapter 6) matters even more in Python. Book: Section 2.3.5.

Installing Eclipse

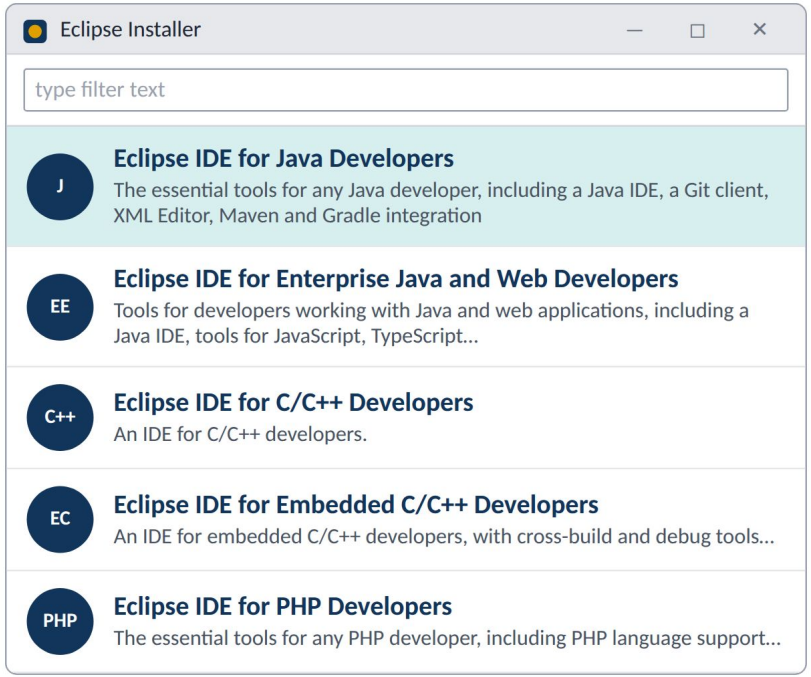


Figure 2.2 The Eclipse Installer's list of packages. Eclipse IDE for Java Developers is the one to choose.

*From `eclipse.org/downloads`, the Eclipse Installer: choose **Eclipse IDE for Java Developers** (Figure 2.2), keep the offered Java runtime, choose a folder, **INSTALL**, then **LAUNCH**. The Java Eclipse runs on and the JDK your projects use are separate settings. Book: Section 2.4.1 and Figures 2.2–2.3.*

The Workspace

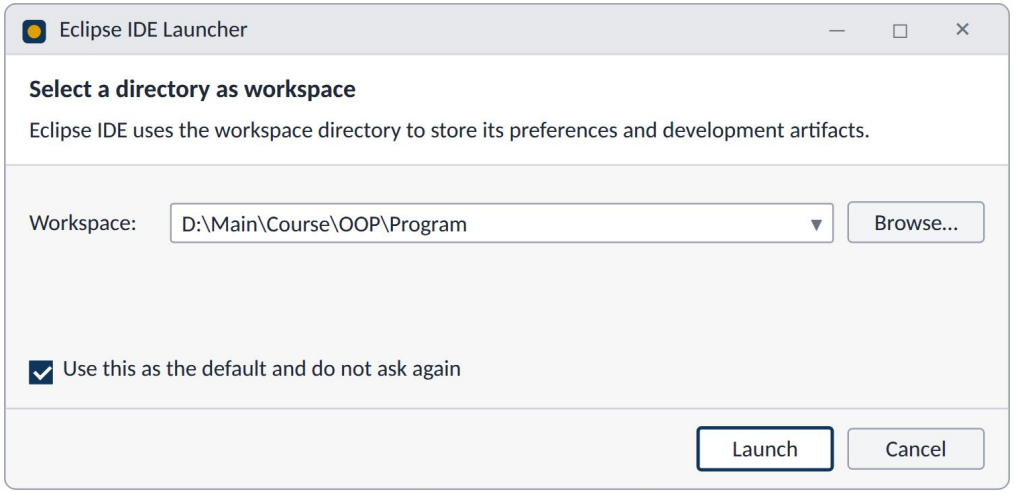


Figure 2.3 The Eclipse IDE Launcher asks for a workspace folder before the workbench opens.

*An Eclipse **workspace** (Figure 2.3) is a folder holding Eclipse's metadata (in `.metadata`) and, by default, your projects. One workspace per course keeps settings apart; File > Switch Workspace changes it. Book: Section 2.4.2 and Figures 2.4–2.5.*

Views, Editors, and Perspectives

> **DEFINITION.** A *view* is a panel that shows structured information and lets you act on it directly. An *editor* shows one file for you to change; changes are not written to disk until you save. A *perspective* is a named arrangement of views and editors suited to one task.

Renaming a file in a view renames it on disk at once; typing in an editor changes nothing until **Ctrl+S**. A **dirty** editor shows `*MyFirstClass.java` — and *Run* runs the last *saved* version.

Two perspectives do almost all the work: **Java** and **Debug**. *Window > Perspective > Reset Perspective...* rescues a messed-up window.

Book: Sections 2.5.1–2.5.2 and Figures 2.7–2.8.

The Java Perspective

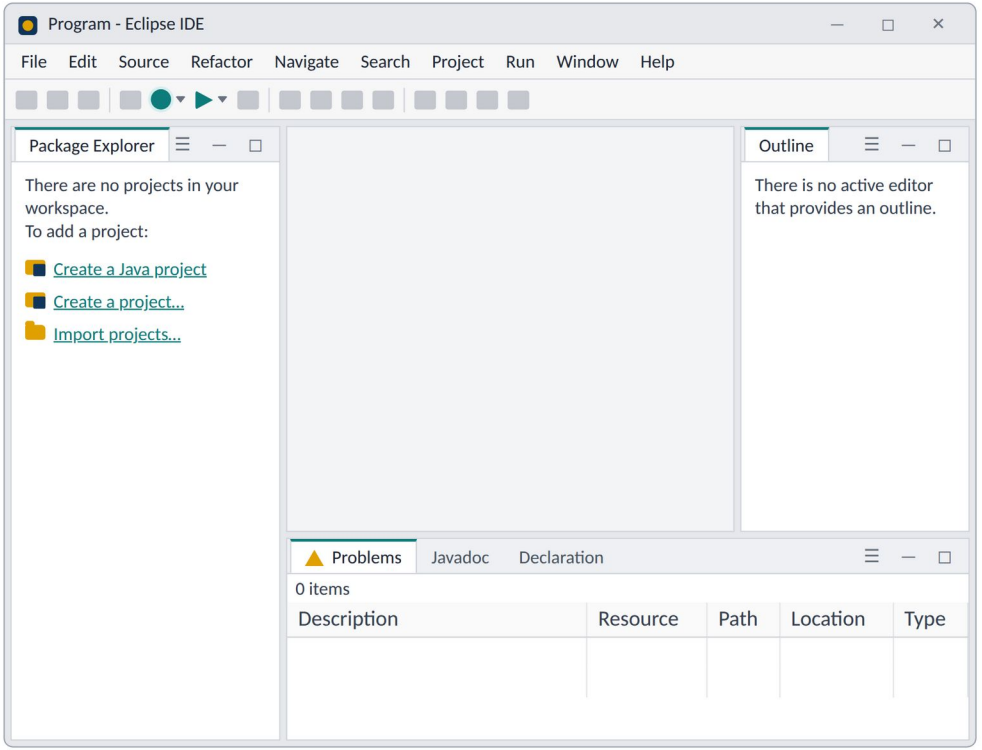


Figure 2.4 The Java perspective of an empty workspace: Package Explorer on the left, editor area in the middle, Outline on the right, Problems, Javadoc and Declaration along the bottom.

In Figure 2.4: Package Explorer, editor area, Outline; Problems, Javadoc and Declaration below. Turn on Show Line Numbers now. Book: Sections 2.5.3–2.5.4 and Figure 2.9.

A Project, a Package, a Class

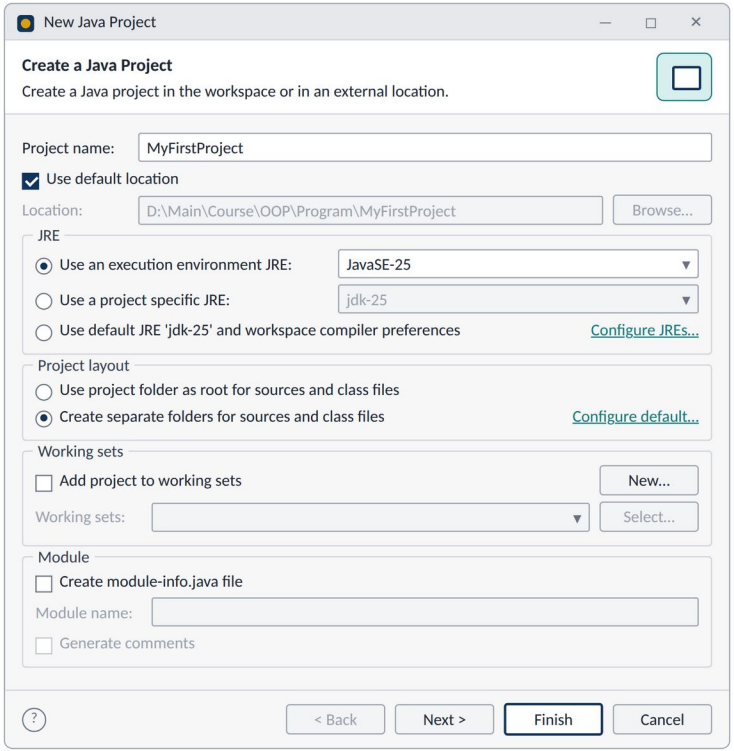


Figure 2.5 The New Java Project wizard, with separate folders for sources and class files.

File > New > Java Project, name `MyFirstProject`, keep separate `src` and `bin` folders (Figure 2.5); then New > Package `myfirstproject` (lowercase), then New > Class `MyFirstClass` with the `main` stub. Book: Sections 2.6.1–2.6.3 and Figures 2.14–2.17.

The First Program of the Project

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

```java
package myfirstproject;

public class MyFirstClass {

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

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

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


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


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

The `package` line of Listing 2.3 must match its folder; `if __name__ == "__main__":` in Listing 2.4 runs `main()` only when the file is run, not imported. Book: Section 2.6.3 and Listings 2.3–2.4.

Running Outside Eclipse: the JAR

A **JAR** (Java ARchive) is a ZIP file of `.class` files plus a *manifest*. *Export… > Java > JAR file*, choose `myprogram1.jar`, **Finish**.

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

```
Hello Eclipse!
```

The **classpath** is where the JVM looks for classes; the class is named *with its package*. The package has become a folder inside the JAR.

`jar tf myprogram1.jar` *lists the contents; a Runnable JAR runs with* `java -jar myprogram1.jar`*. Book: Section 2.6.4 and Figures 2.20–2.22.*

The Same Work in VS Code

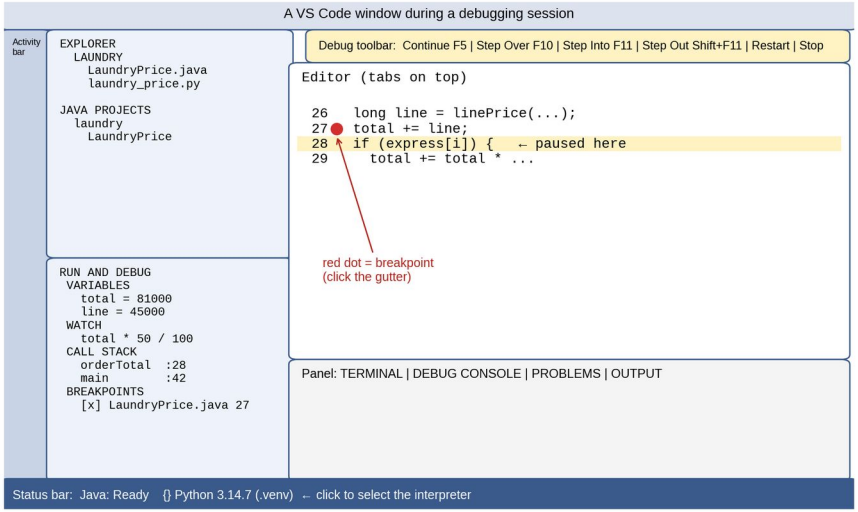


Figure 2.6 A map of the VS Code window while debugging: Explorer and Java Projects in the side bar, the editor with a breakpoint in the gutter, the Run and Debug views, the debug toolbar, the panel, and the interpreter selector in the status bar.

*Figure 2.6 is the map for the rest of today. VS Code opens a **folder** instead of a workspace; settings live in `.vscode`; the **Java Projects** view plays the Package Explorer; **Run** and **Debug** CodeLens links sit above `main`. Book: Sections 2.7–2.7.2 and Figure 2.23.*

Which Python? And a Saved Way to Run

A computer often has several Pythons. The **status bar** shows the selected interpreter; *Python: Select Interpreter* changes it; *Python: Create Environment* makes a `.venv`.

A **launch configuration** in `.vscode/launch.json` stores arguments, a working folder, or a module to run with `python -m`:

```
{
    "type": "java",
    "name": "Tester (Java)",
    "request": "launch",
    "mainClass": "main.Tester"
},
```

F5 starts the selected configuration under the debugger, Ctrl+F5 without it; `debugpy` is the Python debugger's type. The Terminal panel always shows the exact command that ran. Book: Sections 2.7.3–2.7.5.

Packages Are Folders

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.

```
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
```

Full names prevent collisions: `util.Counter` *and the library's* `org.w3c.dom.css.Counter` *are different classes. Book: Sections 2.8.1–2.8.3.*

`Counter`: Package, Field, Getter

Listing 2.5 `Counter.java`, part 1: the class in package `util`, with one field and its getter.

```
package util;

public class Counter {

    private int result = 0;

    public int getResult() {
        return result;
    }
```

`package util;` must match the folder `util` below `src`. The field `result` belongs to the object and survives between calls — Exercise 2.7 shows why that matters.

The template version of `count` holds only `TODO` comments and `return 0;`; Eclipse collects every `TODO` in its *Tasks* view.

Listing 2.5 opens the class that `Tester` will check. Book: Section 2.8.3, Listing 2.5 and Figure 2.24.

`Counter.count`, and What It Must Return

Listing 2.6 `Counter.java`, part 2: `count` adds $1+2+\cdots+x$ and throws an exception outside $1 \le x \le 255$.

```
    // 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;
    }
}
```

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

BY HAND. $S(5) = 1+2+3+4+5 = 15 = \frac{5 \cdot 6}{2}$. Equation (2.1) gives the value `Tester` checks.

Listing 2.6 refuses bad input by throwing an exception (Chapter 5). Book: Section 2.8.3, Listing 2.5 and Equation (2.1).

`Tester`: a Class in Package `main`

Listing 2.7 `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");
        }
    }
}
```

`import util.Counter;` *lets Listing 2.7 use the short name. It prints* `Correct` *and* `Works as expected`*. Book: Section 2.8.3 and Listing 2.6.*

The Python Twins

Listing 2.8 `util/counter.py`, the method `count`: the refusal is a `ValueError`.

```
    # 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 2.9 `main/tester.py`, its first lines: the import names the module, then the thing inside it.

```
"""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)
```

Listing 2.8 and Listing 2.9: run with `cd python` and `python3 -m main.tester`, from the folder above the packages; `python3 main/tester.py` fails with `ModuleNotFoundError`. Book: Section 2.8.3 and Listings 2.7–2.8.

The Missing Import, in Both Languages

```
src/main/Tester.java:7: error: cannot find symbol
        Counter counter = new Counter();
        ^
  symbol:   class Counter
  location: class Tester
  File "/home/student/counter/py/main/tester.py", line 5, in main
    counter = Counter()
              ^^^^^^^
NameError: name 'Counter' is not defined. Did you mean: 'counter'?
```

Java: `cannot find symbol` at *compile time* — nothing runs. **Python:** `NameError` at *run time*, only when line 5 is reached.

The Java message appears twice, once for the type and once for the constructor. The `<frozen runpy>` frames of the full Python traceback belong to `python -m`. Book: Section 2.8.4.

The IDE Offers the Repair

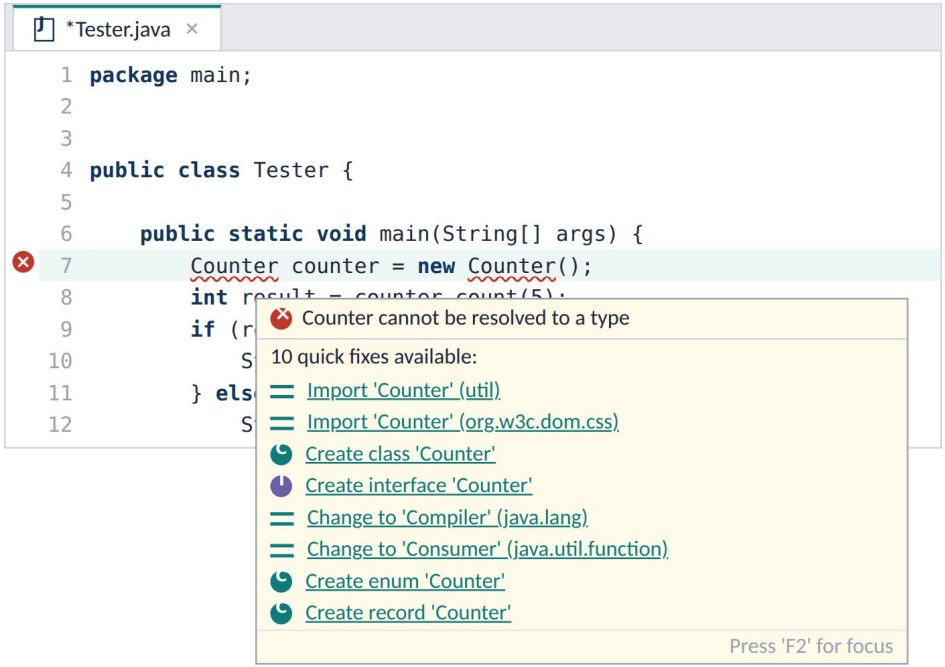


Figure 2.7 Counter cannot be resolved to a type: the quick-fix box offers to import util.Counter, or the unrelated org.w3c.dom.css.Counter.

In Figure 2.7 the first quick fix adds `import util.Counter;`; the second offers the library's unrelated `Counter` — the name collision packages exist to solve. Source > Organize Imports (Ctrl+Shift+O) adds and sorts all imports at once. Book: Section 2.8.4 and Figures 2.25–2.26.

Compiling Packages by Hand

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

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

`-d bin` writes one subfolder per package; `-cp bin` puts `bin` on the classpath; the class is named with dots, `main.Tester` — never a slash, never `.class`.

These two lines are exactly what Eclipse and VS Code do when you press Run. To share whole projects: Export > Archive File and Import > Existing Projects; in VS Code a project is just a folder. Book: Sections 2.8.5–2.8.6.

Editing Power Tools

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. Eclipse: F11 starts debugging, F5 steps into. VS Code: the reverse.

Content assist, Quick Fix, Rename, Format: Table 2.3. Book: Section 2.9 and Table 2.3.

Debugging from Zero

A **bug** is a difference between what a program does and what it should do. Loud bugs stop with a message; silent bugs print a believable wrong number — Rp 109,350 looks as real as Rp 93,150.

Print debugging works anywhere, but every question means edit and rerun. The **debugger** pauses at any line and shows every variable without editing.

> **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.

Book: Section 2.10.1.

Seven Words for Any Debugger

Breakpoint: stop *before* running this line. **Suspended:** stopped, line highlighted. **Stepping:** step over, into, out. **Resume:** run to the next breakpoint.

Variables of the current method; a **watch** such as `total * 50 / 100`, re-evaluated at every stop.

The **call stack**: the methods currently running, each called by the one below it.

A **watchpoint** stops whenever a particular *field* is read or modified, whichever line does it.

Book: Section 2.10.2.

A Breakpoint in Eclipse

Tester.java × Counter.java

```
1  package main;
2
3  import util.Counter;
4
5  public class Tester {
6
7      public static void main(String[] args) {
8          Counter counter = new Counter();
9          int result = counter.count(5);
10         if (result == 15) {
11             System.out.println("Correct");
12         } else {
13             System.out.println("False");
14         }
15         try {
16             counter.count(256);
17         } catch (RuntimeException e) {
18             System.out.println("Works as expected");
19         }
20     }
21 }
```

Figure 2.8 A breakpoint on line 8, Counter counter = new Counter(); the dot in the ruler marks it.

Double-click the ruler, Toggle Breakpoint, or Ctrl+Shift+B (Figure 2.8); then Debug As > Java Application. Under Run As breakpoints are ignored. Eclipse offers to switch to the Debug perspective. Book: Section 2.10.3 and Figures 2.30–2.34.

Resume, Step Over, Step Into, Step Out

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`

Table 2.4: over what you trust, into what you suspect. Book: Section 2.10.4 and Table 2.4.

Stepping, Seen on the Call Stack

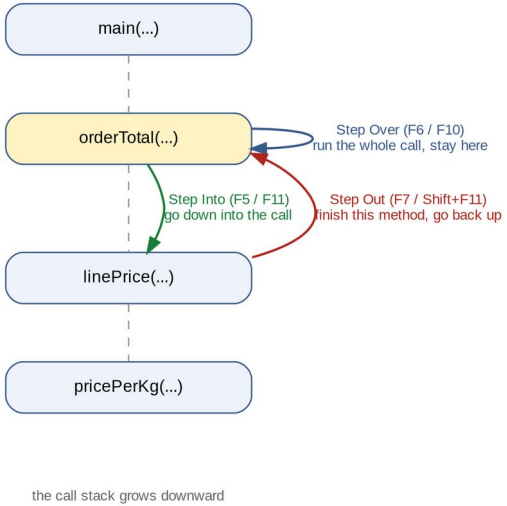


Figure 2.9 Stepping seen on the call stack. Step Into goes down into the called method, Step Out returns to the caller, and Step Over runs the whole call and stays at the same level.

In Figure 2.9, from a line of `orderTotal` that calls `linePrice`, which calls `pricePerKg`: Step Over stays at the same level, Step Into goes down, Step Out climbs back to the caller. Book: Section 2.10.4 and Figure 2.36.

The Call Stack and the Variables

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`.

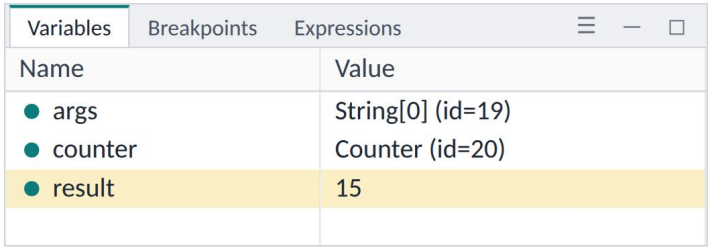


Figure 2.10 The Variables view after count(5) returned: args, counter and result = 15.

Click a lower frame to see its line and variables. The Variables view (Figure 2.10) highlights what the last step changed; you can even change a value to ask "what if?". Book: Sections 2.10.5–2.10.6 and Figures 2.37–2.38.

Managing Breakpoints

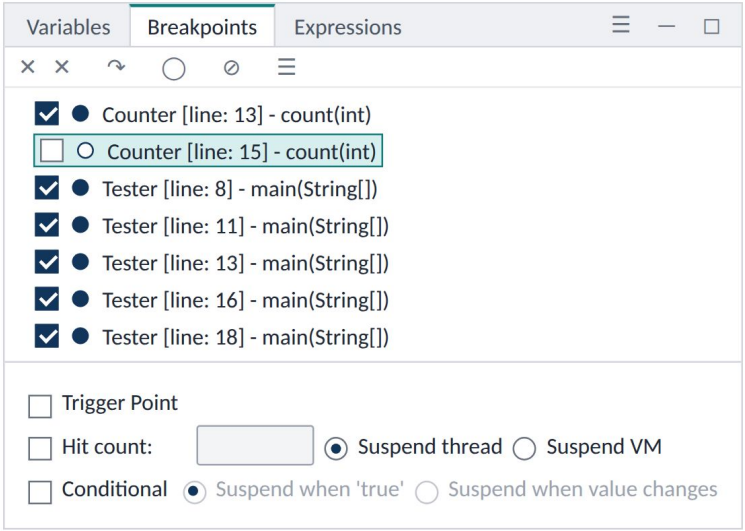


Figure 2.11 The Breakpoints view lists every breakpoint; the check boxes enable or disable them, and the lower part holds the Hit count and Conditional properties.

The Breakpoints view (Figure 2.11) disables without deleting and holds two powerful properties: ***Hit count*** *(stop on the* n*-th arrival) and* ***Conditional*** *(stop when* `i == 7` *is true). Skip All Breakpoints (Ctrl+Alt+B) silences them all. Book: Section 2.10.7 and Figures 2.42–2.43.*

The Same Debugger in VS Code

Click the gutter (or F9) for a red dot; **Debug** CodeLens or F5 to start. The Run and Debug side bar shows **Variables**, **Watch**, **Call Stack**, **Breakpoints**.

Toolbar: Continue (F5), Step Over (F10), Step Into (F11), Step Out (Shift+F11), Restart, Stop. The **Debug Console** evaluates any expression.

Add Conditional Breakpoint…: Expression, Hit Count, or **Log Message** — a logpoint prints without stopping.

> **KEY INSIGHT.** Every graphical debugger offers the same small set of controls. Learn the ideas once, on one tool, and you can use any debugger within minutes.

Book: Section 2.10.8.

Worked Hunt: the Bill That Was Too High

wash Rp 6,000/kg, *iron* Rp 4,000/kg, *wash+iron* Rp 9,000/kg; under 3 kg charged as 3 kg; *express* +50% on that item; members 10% off a bill of at least Rp 100,000.

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

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

Bu Sari, a member: 4 kg wash, 2 kg iron, 5 kg wash+iron express. The program says **Rp 109,350**; her receipt says **Rp 93,150**.

Equation (2.2) prices one item and Equation (2.3) the bill. Book: Section 2.11.1 and Equations (2.2)–(2.3).

The Price Calculator: Prices per Line

Listing 2.10 `LaundryPrice.java`, part 1: the constants, the price list as a `switch`, and the price of one line.

```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));
    }
```

Listing 2.10: `static final` constants and the 3 kg minimum. Book: Section 2.11.1 and Listing 2.9.

The Price Calculator: the Total, with a Bug

Listing 2.11 `LaundryPrice.java`, part 2: `orderTotal`, where all item prices come together. This version has a bug.

```
    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;
    }
```

```
Customer : Bu Sari (member)
Total    : Rp 109350
Expected : Rp 93150
```

`main` passes the three items of Listing 2.11 and prints the total beside the expected one. The Python version prints the same wrong total. Book: Section 2.11.1 and Listings 2.9–2.10.

Step 1: Know What the Answer Should Be

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

Table 2.5 gives more than the answer: the running total should pass through 24,000, then 36,000, then 103,500. The first place the program departs from this column is the bug. Book: Section 2.11.2 and Table 2.5.

Steps 2–3: a Hypothesis and a Guided Tour

```
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]);
```

Too high by Rp 16,200: the discount is not missing (that would give 103,500), so look where prices come together — a breakpoint on line 26. In `jdb`: `step` into `pricePerKg`, `where` shows four frames, two `step up` climb back. Item 1 is priced correctly: `line = 24000`. Book: Sections 2.11.3–2.11.4.

Step 4: Watch the Running Total

Breakpoint on `total += line;`, watches on `total` and `total * EXPRESS_PERCENT / 100`, then Resume from item to item.

Stop 1: `line = 24000`, `total = 0`. Stop 2: `line = 12000`, `total = 24000`. Both correct.

Stop 3: `total = 81000` after adding 45,000; the watch shows a surcharge of $81,000 \times 50/100 = 40,500$, but Table 2.5 says 22,500. One step later `total` is **121,500**, not 103,500.

> **THE EXPLANATION.** The surcharge multiplies the running total of *all* items: items 1 and 2 paid $36,000 \times 50/100 = 18,000$ too much, and $18,000 \times 90/100 = 16,200$ after the discount — exactly the overcharge.

Explained to the last rupiah: that is how you know it is the bug and not merely a bug. Book: Section 2.11.5.

The Bug as a Formula

$$t_i = (t_{i-1} + p'_i)\left(1 + \frac{e_i}{2}\right), \tag{2.4}$$

Here $p'_i = r(s_i)\max(w_i, 3)$, while 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 is applied to the wrong thing.

Equation (2.4) also says when the bug is invisible: if the only express item comes *first*, $t_{i-1} = 0$ and the two agree. A test with a single express item never catches it.

Book: Section 2.11.5 and Equation (2.4).

The Same Hunt in `pdb`, with `display`

```
(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
```

`display` plays the watch: at every stop `pdb` reprints each displayed expression whose value changed, with its old value. The last line says it all: `121500  [old: 81000]`. The `jdb` version of this hunt is in the book. Book: Section 2.11.5.

Step 5: Fix, and Check

Listing 2.12 `LaundryPriceFixed.java`, the only changed method: the surcharge now multiplies `line`, and `total += line` moves after it.

```
    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;
    }
```

```
Customer : Bu Sari (member)
Total    : Rp 93150
Expected : Rp 93150
```

With Listing 2.12 both languages print Rp 93,150. Run the fix under the debugger once more and confirm 24,000, 36,000 and 103,500: a right total for the wrong reason is a bug waiting for its next customer. Book: Section 2.11.6 and Listings 2.11–2.12.

The Fix in Python

Listing 2.13 `laundry_price_fixed.py`, the corrected `order_total`.

```
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
```

Same one-line change: the surcharge on `line`, with `//` for whole rupiah, then `total += line`.

Listing 2.13 prints `Total     : Rp 93150`*. Chapter 6 turns such checks into automatic tests. Book: Section 2.11.6 and Listing 2.12.*

The Method Behind the Hunt

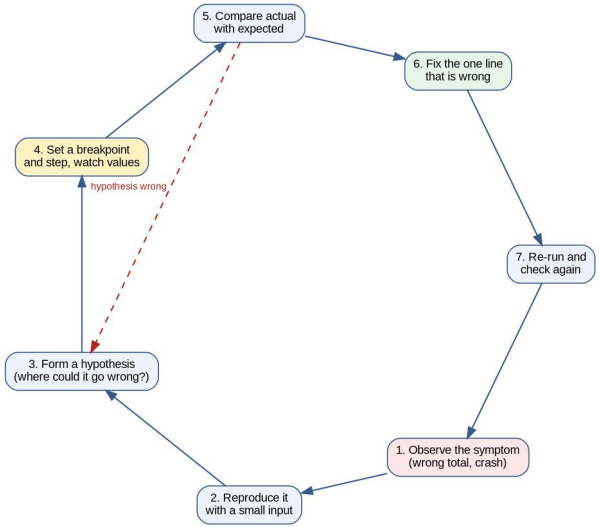


Figure 2.12 The debugging cycle. When the comparison in step 5 refutes the hypothesis, go back to step 3 with what you have learned, not back to guessing.

Figure 2.12: observe, reproduce with the smallest input, hypothesize where to look, break and step, compare with values worked out by hand, fix the one wrong line, re-run old and new cases. A refuted hypothesis is progress: part of the program is ruled out. Book: Section 2.11.7 and Figure 2.44.

Worked Hunt: the Day That Was Rp 6,000 Short

Listing 2.14 `DailyOrders.java`, the method `linePrice`: ten orders run through it, and the day's total is Rp 6,000 short.

```
    static long linePrice(String service, double kg) {
        long billedKg = (long) Math.max(kg, MIN_KG);
        return billedKg * pricePerKg(service);
    }
```

```
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
```

The billed weight must never be less than the real weight, so Listing 2.14 gets a conditional breakpoint `billedKg < kg` on its `return`. Book: Section 2.12 and Listings 2.13–2.14.

Only Two Stops

```
(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)
```

7.5 kg billed as 7, 3.5 kg billed as 3: $0.5 \times 6{,}000 + 0.5 \times 6{,}000 = 6{,}000$. The cast *truncates* the weight; convert the price, not the weight.

Order 9 (1.5 kg) did not stop: the minimum raised it to 3.0 first. Book: Section 2.12.

Reading a Java Stack Trace

Listing 2.15 `UnknownService.java`, its `main`: an order names *dry-clean*, which the price list does not know.

```
    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)
```

Read **from the top**: uncaught, in thread `main`; the *type* `IllegalArgumentException`; the *message*; then the calls, **most recent first** — line 11 threw, called from line 17, called from line 24.

The line that threw is not always the line that is wrong: `pricePerKg` rightly refused, and the real question is why `main` asked. In Listing 2.15 and its Python twin, look for the frames that name your own code. Book: Sections 2.13–2.13.1 and Listing 2.15.

Reading a Python Traceback

```
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
```

`Traceback (most recent call last)`: read **from the bottom** — type and message, then the innermost frame (line 13), then its callers. The carets point at the exact call.

> **KEY INSIGHT.** A stack trace is the call stack printed at the moment of the crash: exception type, message, chain of calls. Read all three before you change a line.

With an exception breakpoint, or `pdb` in post-mortem mode, the program stops at the `raise` with every frame still inspectable. Book: Section 2.13.2 and Listing 2.16.

Profitina in Practice

Reproduce before you fix: a score that “looks wrong” is first reproduced on a test copy, as in step 2 of Figure 2.12; a fixed bug becomes an automated test, and well over a thousand tests gate every release.

Stack traces are data: a web server cannot stop at a breakpoint in front of users, so it logs tracebacks — without personal data, in line with UU PDP 27/2022 — to be read bottom-up.

Know which machine runs what: which interpreter, which model (Qwen2.5 locally through Ollama), which configuration. A health endpoint reporting the deployed version is the status bar of production.

The app advises and the user decides; debugging makes the advice correct, never a guarantee. Book: Section 2.14.

Common Mistakes

Running the old program (unsaved editor, or no `javac`). **The wrong Python** — check the status bar.

Running a package module as a script — use `python3 -m main.tester`. **Package and folder disagree** — create and move classes through the IDE.

Run instead of Debug; `java main/Tester.class` instead of `java -cp bin main.Tester`.

Stepping into everything; **stopping at the first wrong value without explaining it** to the last rupiah.

Reading the trace from the wrong end — Java newest first, Python newest last. **Leaving debug prints behind** — prefer watches and logpoints.

Book: Section 2.15.

Summary — Five Things to Remember

1. An IDE integrates editing, building, running, debugging and navigation; Table 2.2 translates between Eclipse and VS Code.

2. Java compiles once to bytecode and the JVM runs it; Python compiles inside the interpreter at every start. Java finds missing names at compile time, Python at run time.

3. A package is a folder; run `java -cp bin package.Class` and `python -m package.module`.

4. Breakpoints, step over/into/out, variables, watches, the call stack; conditional, hit-count and exception breakpoints stop only where it matters.

5. Debug by method (Figure 2.12); the laundry bill was too high because the surcharge multiplied the running total (Equation (2.4)).

Book: Section 2.16.

Next Time

Chapter 3 — Classes, Objects, Strings, Types, Conditionals, and Loops.

Every part of a class — fields, constructor, `this` and `self`, getters, setters, `toString` — and memory pictures of references.

Strings, the eight primitive types, three arithmetic traps, `if` and `switch`, loops and tracing tables; Profitina Laundry's `Customer` and `Service`.

Before we meet: Exercises 2.2, 2.5, 2.7 and 2.9 — when errors are caught, a stepping path, a counter that keeps counting, and the bug that hides when express comes first. Solutions are in Section 2.18.

Book: Sections 2.17–2.18, then Chapter 3.

Object-Oriented Programming (EF234302)

Lecture 3 — Classes, Objects, Strings, Types, Conditionals, and Loops

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*, Chapter 3

Lecturer: Irfan Subakti — `yifana@gmail.com`

Department of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya

Today we open a program up and lay every part out on the workbench, then build Profitina Laundry's first two classes. Book: Chapter 3.

Agenda

1. Inside a class: fields, constructor, methods, `this` and `self`; creating objects; checking that a class works.

2. References and memory pictures: values, arrows, aliases, identity and equality.

3. Strings and types: positions, immutability, formatting; the eight primitive types; static and dynamic typing.

4. Three arithmetic pitfalls — overflow, floating point, integer division — then `if`, `switch`, loops and tracing tables.

5. Worked exercises, and Profitina Laundry's `Customer` and `Service`, checked by hand.

Book: Chapter 3, Sections 3.1–3.17.

Where We Left Off

Chapter 1 set up the workbench; Chapter 2 made it comfortable — projects, packages, and the debugger. The programs themselves have been tiny.

Two groups of parts: **structure** — classes and objects; and **what happens inside a method** — values, decisions, repetition.

> **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.

A programmer who has not mastered the second group writes beautiful classes that compute the wrong answer. Book: Section 3.1.

Hello, Once More, Word by Word

Listing 3.1 `Hello.java`: the smallest complete Java program.

```java
public class Hello {

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

Listing 3.2 `hello.py`: the same program in Python, with the conventional `main` function.

```python
def main():
    print("Hello world!")


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

*In **Listing 3.1** all code lives in a class, braces mark blocks, and `S`, dots, quotes and the semicolon are all required. **Listing 3.2** marks blocks by **indentation** and calls `main` only when the file is run directly. Book: Section 3.2.1 and Listings 3.1–3.2.*

Reading Words from the Command Line

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!");
        }
    }
}
```

```
java Hello2 Welcome to Profitina Laundry
```

```
She said, Welcome!
```

*In **Listing 3.3**, `args[0]` is the first word — numbering starts at 0 — and `+` **concatenates**. Without the `args.length > 0` test, no words would mean an `ArrayIndexOutOfBoundsException`. In Python the first word is `sys.argv[1]`, because `sys.argv[0]` is the script's name. Book: Section 3.2.2 and Listing 3.3.*

Names, Conventions, and Comments

Java: classes `LaundryDemo`, variables and methods in **camel case** `aLineOfText`, constants `MAX_VALUE`. **Python** (PEP 8): classes `LaundryService`, variables and functions in **snake case** `a_line_of_text`.

Rules, not conventions: `String`, not `string`; `print`, not `Print`.

Comments: line (`//`, `#`), block (`/* … */`), and documentation (Javadoc `/** … */`, Python docstring). Comment *why*, not *what*:

```
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 above repeats the code and adds nothing; "members get 10% off" is a good comment. Book: Sections 3.2.3–3.2.4 and Listings 3.5–3.6.

Blueprints and Houses

A **class** is a blueprint; an **object** is a house built from it — a real thing with its own furniture. *Object* and *instance* mean the same.

A class `Dog` says every dog has a name, a breed, an age and a weight. Fred and Bella are two instances.

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**: the header row is the class, each row below one object. Book: Section 3.3.1 and Table 3.1.*

The Class and Two Objects in UML

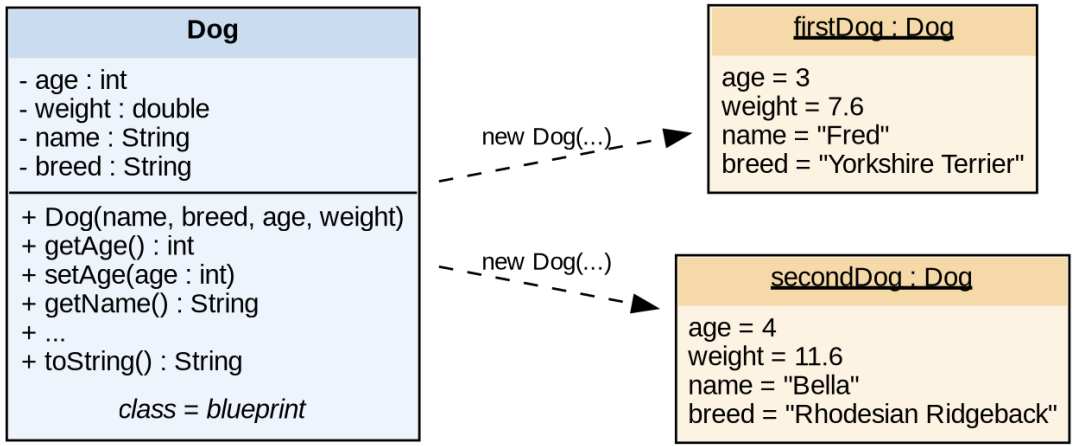


Figure 3.1 The class `Dog` as a blueprint (left) and two objects built from it with `new` (right); each object has its own copy of every field.

*The most important thing in **Figure 3.1** is easy to miss: each object has its own copy of every field. Fred's `age` and Bella's `age` are two different boxes. Book: Section 3.3.1 and Figure 3.3.*

`Dog.java`: Fields and Constructor

Listing 3.4 `Dog.java`, part 1: four private fields and one constructor.

```
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;
    }
```

Fields hold each object's data; `private` hides them behind methods — **encapsulation**. The **constructor** has the class's name and no return type; its **parameters** receive the caller's values. No `main`: `Dog` is a blueprint, not a program.

Listing 3.4 *opens the complete class of the book. Book: Section 3.3.2 and Listing 3.7.*

`Dog.java`: a Getter, a Setter, and `toString`

Listing 3.5 `Dog.java`, part 2: the getter and setter for `age`; weight, name and breed follow the same pattern.

```
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
```

Listing 3.6 `Dog.java`, part 3: `toString`, which describes the whole dog as text.

```
    @Override
    public String toString() {
        return "Dog name: " + getName() + "; breed: " + getBreed()
                + "; age: " + getAge() + "; weight: " + getWeight() + " kg";
    }
}
```

*A **getter** returns one field, a **setter** changes one — and is a gate where tomorrow's `if (age < 0) return;` protects every caller at once (**Listing 3.5**). `@Override` in **Listing 3.6** catches a misspelled `tostring`. Book: Sections 3.3.2 and 3.3.4 and Listing 3.7.*

The Parts of a Class, Side by Side

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

***Table 3.2**: the Java parts and their Python counterparts. Book: Section 3.3.2 and Table 3.2.*

`dog.py`: the Same Class in Python

Listing 3.7 `dog.py`, part 1: fields are created inside `__init__`; a getter and a setter.

```python
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
```

*In **Listing 3.7**: no field declarations, no `private` (`_age` is a convention), no types; `__str__` plays `toString`. Book: Sections 3.3.2 and 3.3.4 and Listing 3.8.*

The Word `this`, and the Word `self`

```
public Dog(String name, String breed, int age, double weight) {
    this.name = name;
    ...
}
```

Two things are called `name`: the field and the parameter. `name = name;` copies the parameter into itself — a silent bug that leaves the field `null`.

> **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.

Python makes `self` compulsory: `first_dog.get_age()` is really `Dog.get_age(first_dog)`. Book: Section 3.3.3.

Creating Objects with `new`

Listing 3.8 `DogDemo.java`, its `main`: two `Dog` objects created, printed, and changed through setters.

```
    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
    }
```

```
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
```

*`new` calls the constructor with the **arguments** and returns a **reference**; Fred's change in **Listing 3.8** does not touch Bella. Book: Section 3.3.5 and Listing 3.9.*

The Demonstration in Python

Listing 3.9 `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()
```

*No `new`: call the class like a function. Java's `+` turns a number into text; Python demands `str(...)`. The output of **Listing 3.9** is the same four lines. Book: Section 3.3.5 and Listing 3.10.*

Checking That a Class Works

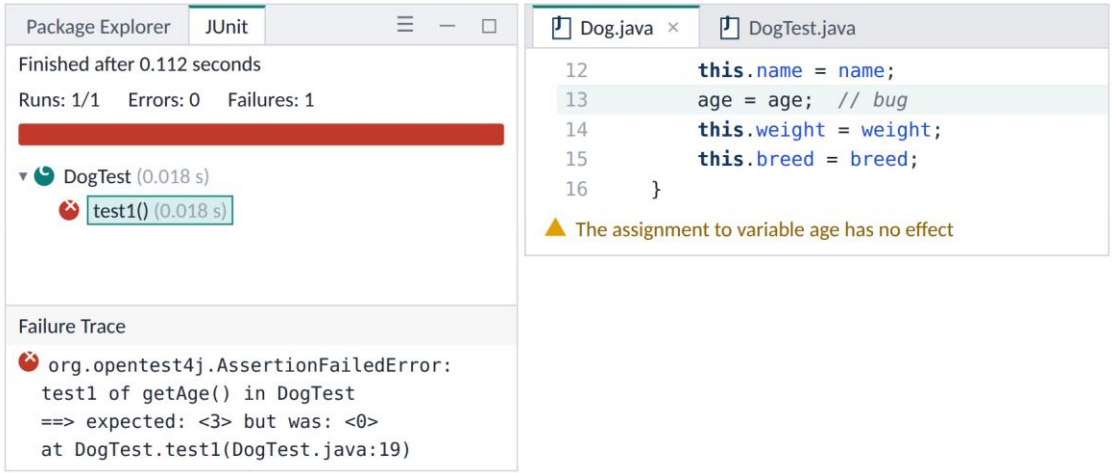


Figure 3.2 A red bar: `test1` fails with expected 3 but was 0, because the constructor of this `Dog` says `age = age;` instead of `this.age = age;`.

*`DogCheck` compares expected and actual and prints PASS or FAIL (weights within 10^{-9}). JUnit does it at scale: in **Figure 3.2** a constructor that says `age = age;` gives `expected: <3> but was: <0>`; with `this.age = age;` the bar turns green. Every test: arrange, act, assert. Book: Section 3.3.6, Listings 3.11–3.12 and Figures 3.6–3.8.*

Values and Arrows

Listing 3.10 `ReferenceDemo.java`, the reference half: an alias, a twin with equal contents, and `null`.

```
        // 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);
    }
```

Primitive variables hold values; **reference** variables hold an arrow to an object on the **heap**. `Dog alias = fido;` copies the *arrow*: two **aliases**, one dog.

Listing 3.10 *begins with* `int b = a; b = 9;`*, which leaves* `a` *alone. Book: Section 3.4.1 and Listing 3.13.*

The Memory Picture

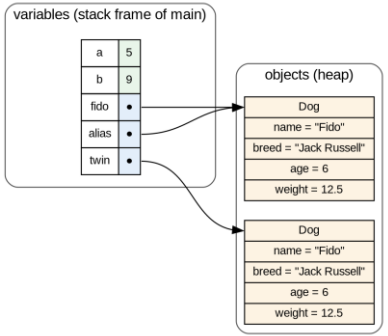


Figure 3.3 Memory inside `main` of Listing 3.13: `a` and `b` hold values, while `fido`, `alias` and `twin` hold arrows; `fido` and `alias` point to the same object.

```
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 3.3**: `==` on references asks "same object?"; calling through `null` gives `NullPointerException`. Book: Section 3.4.1 and Figure 3.10.*

Python: Every Variable Is a Name

Listing 3.11 `reference_demo.py`, an excerpt: `is` tests identity.

```
# 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))
```

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`

Listing 3.11 *prints what Java printed;* ***Table 3.3*** *spells the two questions in opposite ways. Book: Sections 3.4.2–3.4.3, Listing 3.14 and Table 3.3.*

Strings: Positions and Pieces

Listing 3.12 `StringBasics.java`, an excerpt: `substring`, `charAt` and `length`.

```
        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());
```

Listing 3.13 `string_basics.py`, the same excerpt: slices, indexing and `len`.

```
    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)))
```

Include the start, exclude the end: ***Listing 3.12*** *and* ***Listing 3.13*** *print* `Yarik / Hello`, `d`, `5`. *Book: Sections 3.5.1–3.5.2 and Listings 3.15–3.16.*

Everyday String Operations

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()`

Task	Java (`String s`)	Python (`str s`)
Repeat	`s.repeat(3)`	`s * 3`
Same content	`s.equals(t)`	`s == t`

Table 3.4 *is a summary to keep at hand. Book: Section 3.5.2 and Table 3.4.*

Strings Never Change

Listing 3.14 `StringImmutability.java`, the core: methods return new strings; `==` compares arrows, `equals` compares characters.

```
        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));
```

```
s = laundry, t = LAUNDRY
s = laundry day
code == typed:      false
code.equals(typed): true
```

*`toUpperCase` builds a new string, and `s = s + " day"` moves the arrow. For many pieces use `StringBuilder` (Python: `"".join(parts)`). In **Listing 3.14**, `code == typed` is `false`. Book: Section 3.5.3, Listing 3.17 and Figure 3.12.*

Same Characters, Two Objects

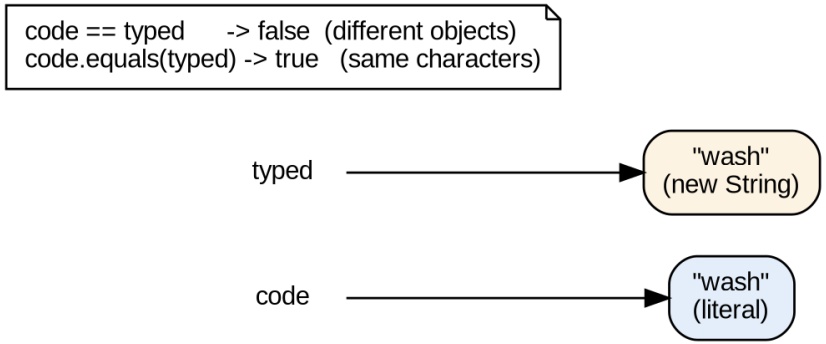


Figure 3.4 Two different `String` objects with the same characters: `==` says `false`, `equals` says `true`.

> **WARNING.** In Java, always compare the *content* of strings with `equals`, never with `==`. `"wash" == "wash"` happens to be `true` for literals, but text typed, read or built at run time is a fresh object.

***Figure 3.4**: Python avoids the trap because its `==` compares content. Book: Section 3.5.3 and Figure 3.13.*

Formatting Output, and Parsing Input

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%`

`printf` prints, `String.format` returns; Python's f-string `f"{weight_kg:.2f}"`. Java's `%,d` follows the computer's **locale**: `24.192` on an Indonesian setting.

Parse text first: `Integer.parseInt`, `Double.parseDouble`; Python `int(...)`, `float(...)`. `"12" + "3"` is `"123"`; `Integer.parseInt("12") + 3` is `15`.

Table 3.5 *lists the placeholders the book uses;* `abc` *fails with* `NumberFormatException` *or* `ValueError`*. Book: Sections 3.5.4–3.5.5 and Table 3.5.*

Java's Eight Primitive Types

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`

$$-2^{b-1} \le v \le 2^{b-1} - 1 \tag{3.1}$$

*By Equation (3.1) a `byte` holds -2^7 to $2^7 - 1$ and an `int` -2^{31} to $2^{31} - 1$. Choose `int`/`long` and `double`; Python has one unlimited `int` and one `float` (**Table 3.6**). Book: Section 3.6.1, Table 3.6 and Equation (3.1).*

Static Typing in Java

Listing 3.15 `StaticTyping.java`: in Java every variable has one type for life; `var` lets the compiler work the type out.

```java
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
    }
}
```

```
StaticTyping.java:11: error: incompatible types: String cannot be converted to int
        count = "three";
                ^
1 error
```

*In **Listing 3.15** an `int` is **widened** to `double`; `var` asks the compiler to infer `String`; `7000L` is a `long`. Uncomment `count = "three";` and the compiler refuses, as shown. Book: Section 3.6.2 and Listing 3.23.*

Dynamic, but Strong, Typing in Python

Listing 3.16 `dynamic_typing.py`, its `main`: a name may be rebound to another type; the type is checked when `+` runs.

```python
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)
```

```
3 int
three str
7000 int
21000 int True
TypeError: can only concatenate str (not "int") to str
```

***Listing 3.16**: a **type hint** is not enforced; checked late, but checked. Book: Section 3.6.2 and Listing 3.24.*

Characters, Truth Values, and Operators

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`

***Table 3.7**: `*` before `+`; Java has no power operator (`Math.pow`), Python writes `**`. A `char` is a number underneath (`'B'` is 66). `&&` and `||` **short-circuit**, so `note != null && note.length() > 0` is safe. Book: Sections 3.6.3–3.7.1 and Table 3.7.*

Pitfall One: Integer Overflow

Listing 3.17 `IntegerTypes.java`, the overflow: none of x, y, z or the answer is near the limit, yet the result is negative.

```
        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));
```

```
x * y / z as int : -294967
x * y as int     : -1474836480
x * y / z as long: 4000000
MAX_VALUE + 1    : -2147483648
```

*In **Listing 3.17** the intermediate product 2×10^{10} exceeds $2^{31} - 1$. The cure: `(long) x * y / z` — cast before the multiplication. Book: Section 3.7.2 and Listing 3.27.*

The Overflow, Worked by Hand

$$\text{stored}(v) = ((v + 2^{31}) \bmod 2^{32}) - 2^{31} \tag{3.2}$$

$2^{32} = 4\,294\,967\,296$; $v + 2^{31} = 20\,000\,000\,000 + 2\,147\,483\,648 = 22\,147\,483\,648$.

That divided by 2^{32} goes 5 times, $5 \times 4\,294\,967\,296 = 21\,474\,836\,480$; remainder 672 647 168.

$672\,647\,168 - 2\,147\,483\,648 = -1\,474\,836\,480$ — the "x * y as int" line. Divided by 5 000, dropping the fraction: $-294\,967$.

> **WARNING.** Overflow is silent. Multiply money in `long`, or use `Math.multiplyExact`. Python's `int` never overflows; its `wrap_int32` imitates Java.

Equation (3.2) keeps the lowest 32 bits, like an odometer rolling over. Book: Section 3.7.2, Equation (3.2) and Listing 3.28.

Pitfall Two: Tenths That Do Not Add Up

Listing 3.18 `FloatingPoint.java`, the core: ten tenths are not one, addition is not associative, and `==` fails.

```
        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
```

0.1 has no exact binary form, just as $\frac{1}{3}$ has no exact decimal one; ***Listing 3.18*** *shows the consequences. Book: Section 3.7.3 and Listing 3.29.*

How the Error Grows

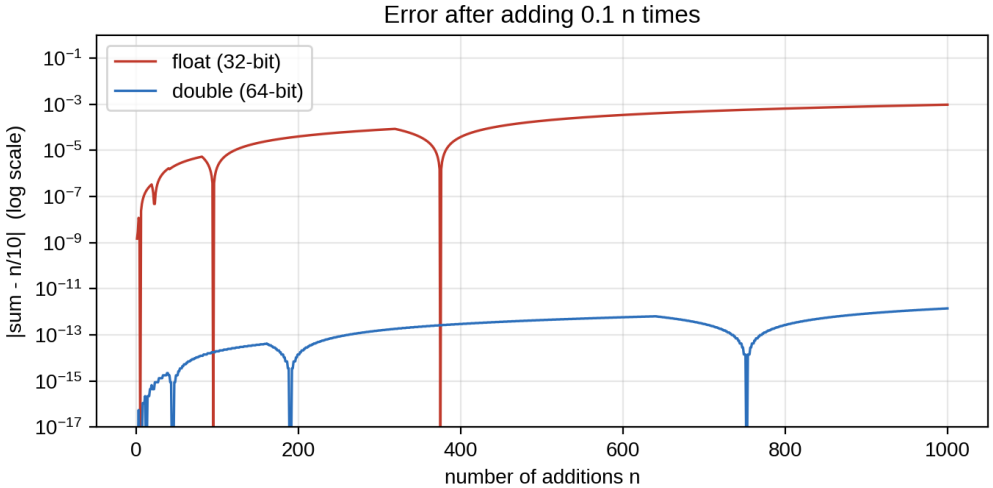


Figure 3.5 The error after adding 0.1 again and again, for `float` and `double`; the dips to the bottom are the moments when the rounding errors happen to cancel.

*In **Figure 3.5** the `float` error is roughly a billion times the `double` error; the dips are moments when rounding errors cancel. Book: Section 3.7.3 and Figure 3.14.*

Compare with a Tolerance

$$|a - b| < \varepsilon \tag{3.3}$$

Java: `Math.abs(a - b) < 1e-9`. Python: the same, or `math.isclose(a, b)`; `Decimal("0.1") * 10` is exactly `1.0`.

> **KEY INSIGHT.** Money should never be stored as `double`. Store whole rupiah in a `long` (Java) or an `int` (Python), as Profitina Laundry's classes do, or use `BigDecimal` / `Decimal`.

Equation (3.3) is exactly what `DogCheck` used for the weight. Book: Section 3.7.3, Equation (3.3) and Listing 3.30.

Pitfall Three: Integer Division

$$a = (a/b) \cdot b + (a \bmod b) \tag{3.4}$$

Listing 3.19 `IntegerDivision.java`, the core: `x / y` between integers, casting before and after, and negative operands.

```
        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));
```

```
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
```

*Equation (3.4) ties `/` and `%` together; `n % 10` is the last digit. `(double) (x / y)` divides first and is too late. In **Listing 3.19** Java truncates toward zero. Book: Section 3.7.4 and Listing 3.31.*

Division and Rounding, Side by Side

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**: Python's `/` always gives a decimal and `//` rounds down, so `-7 // 2` is `-4`; `round(2.5)` is `2` (half to even), while `Math.round(2.5)` is `3`. Both languages obey Equation (3.4). Book: Section 3.7.4 and Table 3.8.*

Doing Something Only When a Condition Holds

Listing 3.20 `IfDemo.java`, part 1: an `if` with an `else`. What does it print, and what if `i` were 1?

```
        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);
        }
```

```
i is less than or equal to 0
j is 3
```

The condition must be `boolean`; exactly one block runs. With `i = 1`: `i is greater than 0!` and `j is 2`. Always write the braces. ***Listing 3.20*** *starts the book's puzzle. Book: Section 3.8.1 and Listing 3.33.*

Chains of Decisions

Listing 3.21 `if_demo.py`, part 2: an `elif` chain tried for $k = 2, \ldots, 6$.

```
    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")
```

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`

*First true condition wins (**Table 3.9**); **Listing 3.21** is the Python chain. Book: Section 3.8.2, Listing 3.34, Table 3.9 and Figure 3.15.*

`switch` and Its Trap

Listing 3.22 `SwitchDemo.java`, part 1: a classic `switch` without `break`.

```
        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");
            }
        }
```

```
-- without break --
Input was b
Input was c
```

*A `case` label is only an entry point: from `case 'b':` execution **falls through** into `case 'c':`. The cure is `break`, plus a `default` for no match (**Listing 3.22**). Book: Section 3.8.3, Listing 3.35 and Figure 3.16.*

The Arrow Form, and Python's `match`

Listing 3.23 `SwitchDemo.java`, part 3: the arrow form (Java 14+), a `switch` that produces a value.

```
        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);
```

Listing 3.24 `switch_demo.py`, `describe`: `match` never falls through; `case _:` is the default.

```
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"
```

*Prefer the arrow form (**Listing 3.23**); **Listing 3.24** needs Python 3.10+. Book: Section 3.8.3 and Listings 3.35–3.36.*

`while`, `for`, and `do-while`

Listing 3.25 `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);
    }
}
```

*Without `i++` the `while` loop in **Listing 3.25** never ends — an **infinite loop**. The `do-while` runs once even though `k < 4` is false. Book: Section 3.9.1 and Listing 3.37.*

The Four Parts of a `for` Loop

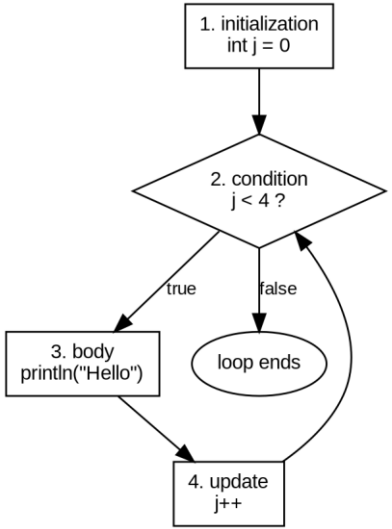


Figure 3.6 The four parts of `for (int j = 0; j < 4; j++)`: initialization once, then condition, body and update in a cycle until the condition is false.

***Figure 3.6**: initialization once; then condition, body, update, until the condition is false — so `j` is 0, 1, 2, 3. Python's `for j in range(4):` does the same; `while True:` with `break` replaces `do`-`while`. Book: Section 3.9.1 and Figure 3.17.*

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

Four sequences: `for (i = 1; i <= 10; i++)`, `for (i = 10; i >= 1; i--)`, `i += 3` up to 15, and `i * i` for squares — Python `range(1, 11)`, `range(10, 0, -1)`, `range(3, 16, 3)`. `i < 10` instead of `i <= 10` is the **off-by-one error**.

***Table 3.10**: pick the loop whose shape tells the reader what kind of repetition this is. Book: Sections 3.9.2–3.9.4, Listings 3.39–3.40 and Table 3.10.*

The For-Each Loop

Listing 3.26 `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();
    }
}
```

```
You gave 3 word(s).
args[0] = wash
args[1] = dry
args[2] = iron
WASH DRY IRON
```

*Run as `java ArgsEcho wash dry iron`. The for-each loop in **Listing 3.26** has no index to get wrong; Python's `for word in args:` is the same, and `enumerate` gives index and value together. Book: Section 3.9.3 and Listing 3.41.*

Tracing Tables

Listing 3.27 `TraceDemo.java`: a `while` loop that takes a number apart digit by digit.

```java
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);
    }
}
```

*`n % 10` is the last digit and `n / 10` drops it (Python `n // 10`). **Listing 3.27** prints `digit sum = 12` and `reversed  = 1704`. Book: Section 3.10 and Listing 3.43.*

Tracing $n = 4071$ by Hand

Table 3.11 Tracing **Listing 3.27** 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

One column per changing variable plus the condition; a starting row; execute *in order*, with values *as they are at that moment*; stop when the condition is false.

***Table 3.11**: one pass per digit; the trailing zero of 4071 becomes a leading zero that disappears. Chapter 4 works whole problems of this kind. Book: Section 3.10 and Table 3.11.*

Worked Exercise: the Area of a Circle

$$A = \pi r^2 \tag{3.5}$$

Listing 3.28 `CircleArea.java`: the area of a circle with radius 5.

```
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);
    }
}
```

*Both are `double`, since π is not whole; **Listing 3.28** prints `78.53981633974483` and, formatted, `78.54` — by hand* $25 \times 3.14159 \ldots$*, as Equation (3.5) says. Book: Section 3.11.1 and Listing 3.45.*

Worked Exercise: Imperial Units to Kilograms

$$W_{\text{lb}} = 2240t + 112h + 28q + 14s + p + \frac{o}{16} + \frac{d}{256} + \frac{g}{7000} \tag{3.6}$$

$$W_{\text{kg}} = 0.45359237 \cdot W_{\text{lb}} \tag{3.7}$$

BY HAND. 11 stones and 6 pounds: $14 \times 11 + 6 = 160$ lb by Equation (3.6); $160 \times 0.45359237 = 72.5747792$ kg by Equation (3.7). The program prints `72.57477920000001`.

Name the factors (`poundsPerStone`), never scatter **magic numbers**; write `1 / 16.0`, because `1 / 16` is 0.

The table of units is Table 3.12; the programs are Listings 3.47–3.48. Book: Section 3.11.2 and Equations (3.6)–(3.7).

Worked Exercise: Five Hundred Years of Interest

$$\text{total} = \text{capital} \cdot (1 + 0.01 \cdot \text{rate})^{\text{years}} \tag{3.8}$$

Table 3.12 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

Equation (3.8): each year multiplies by 1.023, *so n years multiply by* 1.023^n. ***Table 3.12*** *checks the first five years by hand; the repetitive six-statement version is Listing 3.49. Book: Section 3.11.3 and Table 3.13.*

Removing the Repetition with a Loop

Listing 3.29 `InvestmentLoop.java`, its `main`: the balance year by year, printing the first five years and every hundredth.

```
        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);
```

```
 100            971.77
 200           9443.36
 300          91767.76
 400         891771.29
 500        8665963.51
```

*In **Listing 3.29** 500 multiplications agree with one `Math.pow` to the cent: `8665963.51`. Book: Section 3.11.3 and Listing 3.51.*

The Signature of Compound Growth

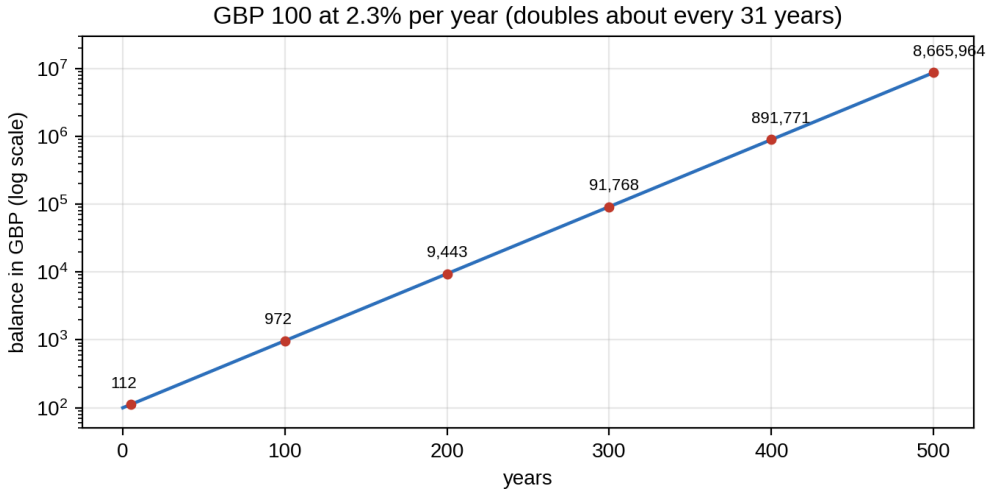


Figure 3.7 The balance of GBP 100 at 2.3% per year over 500 years, on a logarithmic scale; the straight line is the signature of compound growth.

*On the logarithmic axis of **Figure 3.7** growth by the same factor each year is a straight line; the balance doubles about every 31 years. Book: Section 3.11.3 and Figure 3.18.*

Profitina Laundry: `Customer` and `Service`

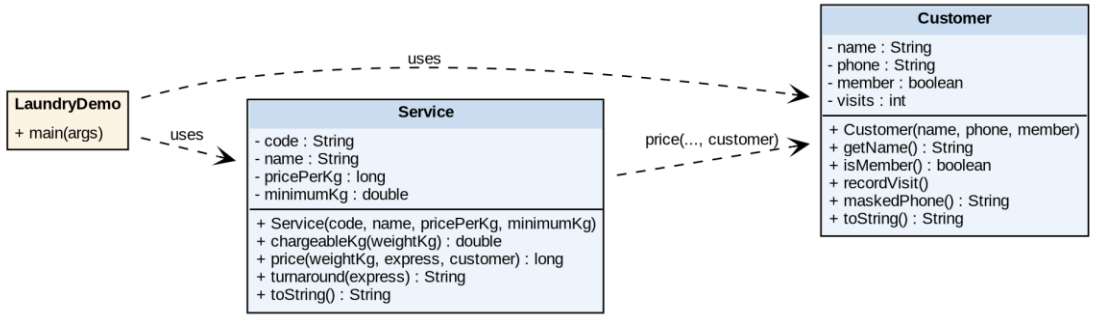


Figure 3.8 The first two classes of Profitina Laundry and the program that uses them.

Wash & Fold Rp7,000/kg, Iron Only Rp5,000/kg, Wash, Dry & Iron Rp10,000/kg; a minimum weight; half-kilogram steps rounded up; express +50%; members 10% off.

*In **Figure 3.8**: name and phone never change, so no setters and `final`; `recordVisit()` instead of a setter; `maskedPhone()` shows three digits. Book: Sections 3.12–3.12.1 and Figure 3.19.*

The Price Rules as Formulas

$$c = \frac{\lceil 2 \cdot \max(w, m) \rceil}{2} \tag{3.9}$$

$$p = \begin{cases} c \cdot r & \text{regular} \\ \frac{3}{2}\, c \cdot r & \text{express} \end{cases}, \qquad \text{price} = \begin{cases} p & \text{guest} \\ p - \lfloor p/10 \rfloor & \text{member} \end{cases} \tag{3.10}$$

BY HAND. $w = 4.2$, $m = 3$: $2w = 8.4$, rounded up 9, halved 4.5 kg by Equation (3.9). At Rp7,000: 31 500; a member pays $31\,500 - 3\,150 = 28\,350$ by Equation (3.10).

After Equation (3.9) every step is exact integer arithmetic in whole rupiah; the floor is integer division. Book: Section 3.12.2, Equations (3.9)–(3.10) and Figure 3.20.

`Service`: Chargeable Weight and Price

Listing 3.30 `Service.java`, the two methods that price a bag: the chargeable weight and the price.

```
    /** 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;
    }
```

*In **Listing 3.30** `price` receives a `Customer` and asks `isMember()` — objects cooperating — and multiplies before dividing in `total * 3 / 2`. Book: Section 3.12.3 and Listing 3.54.*

The Same Two Methods in Python

Listing 3.31 `service.py`, `chargeable_kg` and `price`: `math.ceil`, and `//` wherever Java divides integers.

```
    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
```

`int(...)` instead of Python's `round`, which rounds halves to even; $c \cdot r$ is always whole here, so nothing is lost. The Python output is identical, character for character.

Listing 3.31 *completes the pair;* `Customer` *in both languages is Listings 3.53 and 3.56. Book: Section 3.12.3 and Listing 3.57.*

Four Bags, Checked by Hand

```
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))
```

Table 3.13 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

***Table 3.13**: Budi's 2 kg are charged as 3 kg; 5.75 kg becomes 6 kg; the four prices add up to 178 950. The demonstration program is Listing 3.55. Book: Section 3.12.4 and Table 3.14.*

Profitina in Practice

Classes for the things of the domain: each kind of signal the scanner examines is its own class that scores and explains itself.

Strings everywhere: a web page is text; brand mentions, `robots.txt`, schema.org — all compare *content*, never identity (**Table 3.3**).

Numbers with care: scores compared with the tolerance of Equation (3.3); PROFIT coin amounts as whole smallest units, like rupiah.

Personal data: `maskedPhone` reflects UU PDP 27/2022 — process only what the purpose needs. **Tests as a gate**, well over a thousand, like `DogCheck`.

The platform advises; its users decide. Book: Section 3.13.

Common Mistakes

`name = name;` instead of `this.name = name;` · `==` on Java strings · `name.trim();` alone · believing `Dog b = a;` copies · calling through `null`.

Overflow in the middle — cast *before*: `(long) x * y` · decimals compared with `==` · `1 / 16` is 0 · money in `double`.

A missing `break` · an `else if` chain in the wrong order · a `while` loop that never ends · off-by-one bounds.

`sys.argv[0]` is the script name · comments that repeat the code.

Book: Section 3.14.

Summary — Five Things to Remember

1. A class is a blueprint; objects are built with `new` (or by calling the class), each with its own fields; `this`/`self` is the current object.

2. Primitive variables hold values, reference variables hold arrows; identity is `==`/`is`, equality `equals`/`==`. Draw the object diagram.

3. Strings are immutable and indexed from 0; Java is statically typed, Python dynamically but strongly.

4. Overflow (Equation (3.2)), floating point (compare by Equation (3.3)), integer division — three traps, three habits.

5. `if`/`switch`/`match`, `for`/`while`/for-each; a tracing table shows exactly what a loop does. `Customer` and `Service` price a bag by Equation (3.9) and Equation (3.10).

Book: Section 3.15.

Next Time

Chapter 4 — Evaluation and Discussion I: From a Plain Specification to a Tested Class.

No new features: fourteen complete problems on Chapters 1–3, each solved slowly with the same eight steps, in both languages, with tests.

Fractions, a clock, an account, complex numbers, weights, a fight, rupiah, times, a receipt, names, passwords, traced loops and leap years.

Before we meet: Exercises 3.2, 3.5, 3.6 and 3.10 — references, expressions, a traced loop, and banknotes by division and remainder. Solutions are in Section 3.17.

Book: Sections 3.16–3.17, then Chapter 4.

Object-Oriented Programming (EF234302)

Lecture 4 — Evaluation and Discussion I: From a Plain Specification to a Tested Class

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*, Chapter 4

Lecturer: Irfan Subakti — `yifana@gmail.com`

Department of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya

No new language features today: fourteen complete problems on Chapters 1–3, each solved from nothing, in both languages, with tests. Book: Chapter 4.

Agenda

1. A method for any small programming problem: eight steps, tests first, and how each problem is laid out.

2. Seven classic problems: fraction calculator, clock angle, computer account, complex numbers, weight, fraction class, player and enemy.

3. Seven everyday problems: rupiah, time of day, a laundry receipt, a tidy name, password rules, loops traced by hand, leap years.

4. For each: examples by hand, the design, Java and Python, a *tempting wrong solution*, and a *check yourself*.

5. The fourteen side by side, Profitina, common mistakes, summary.

Before each solution, cover it and try for ten minutes. Book: Sections 4.1–4.22.

Where We Stand

Chapter 1: the workbench, and the central idea — an object bundles data with the operations allowed on it.

Chapter 2: edit, run and debug — breakpoints, stepping, watches. Several tempting wrong solutions today are bugs a debugger finds in a minute.

Chapter 3: classes, fields, constructors, methods; `int`, `long`, `double` and their traps; `String`, `equals`; `if`, `switch`, `while`, `for`.

> **KEY INSIGHT.** Naming the tool a problem needs is already half of the work.

Book: Section 4.1.

Fourteen Problems, One Toolkit

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

Problem	Topic	Main tools from Chapters 1–3
4.14	Leap years	Boolean expressions, `switch`, a method that returns the next object

Table 4.1 *is a map: loops, start with 4.11–4.13; classes, with 4.3 and 4.5. Book: Section 4.1 and Table 4.1.*

How to Approach a Programming Problem

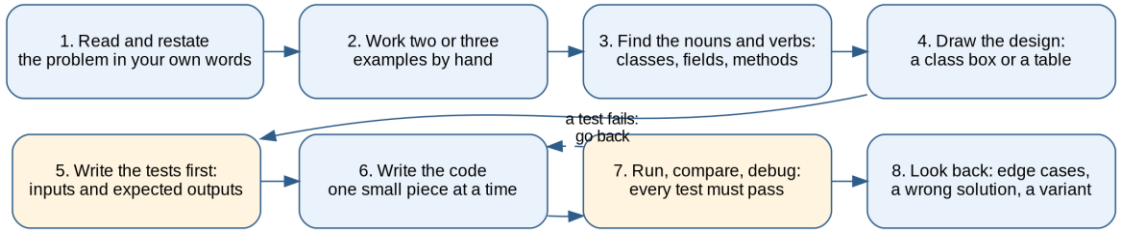


Figure 4.1 The eight steps used for every problem in this chapter; the dashed arrow is the loop between writing code and running the tests.

1 restate in your own words · **2** work two or three examples by hand · **3** nouns → classes and fields, verbs → methods · **4** draw the class box

5 write the tests first · **6** code in small pieces · **7** run, compare, debug · **8** look back: untested input, tempting wrong way, a variant

Figure 4.1 *stretches Pólya's four movements — understand, plan, carry out, look back — into eight steps; the dashed arrow is the loop between code and tests. Book: Sections 4.2–4.2.1 and Figure 4.1.*

Tests, and the Shape of Every Problem

Java: a plain test driver with `check(label, expected, actual)` printing PASS or FAIL — it compares with `equals`, exactly as JUnit's `assertEquals` does.

Python: `unittest` in the standard library; `python3 file.py` runs a demonstration and then the tests.

Each problem: the problem · by hand · classes, fields and methods · Java · Python · line by line · a tempting wrong solution · check yourself.

> **WHY EXAMPLES BY HAND COME FIRST.** A program can only be as right as your understanding of the problem. Every expected value in today's tests was first computed by hand.

Only the `Ran 4 tests in 0.000s` line may differ on your computer. Book: Sections 4.2.2–4.2.3.

Problem 4.1 — Fraction Calculator

PROBLEM 4.1. With `e1`, `d1` and `e2`, `d2` representing $\frac{e_1}{d_1}$ and $\frac{e_2}{d_2}$, write `Q11Fraction` computing `es`, `ds` and `ep`, `dp` for the sum and the product, using Equation (4.1) and Equation (4.2). Test $\frac{1}{2}+\frac{1}{3}, \frac{1}{3}+\frac{3}{4}, \frac{1}{2}\cdot\frac{2}{3}, \frac{1}{4}\cdot\frac{2}{3}$.

$$\frac{e_1}{d_1}+\frac{e_2}{d_2}=\frac{e_1 d_2+e_2 d_1}{d_1 d_2} \tag{4.1}$$

$$\frac{e_1}{d_1}\cdot\frac{e_2}{d_2}=\frac{e_1 e_2}{d_1 d_2} \tag{4.2}$$

By hand: $1\cdot 3+1\cdot 2=5$ over 6; $4+9=13$ over 12; $\frac{2}{6}$; $\frac{2}{12}$ — not reduced, as asked. Edge: $-\frac{1}{2}+\frac{1}{3}=\frac{-1}{6}$.

The answer is a pair of whole numbers, not 0.8333*. Book: Sections 4.3.1–4.3.2 and Equations (4.1)–(4.2).*

4.1: One Formula per Method

Listing 4.1 `Q11Fraction.java`, the four formulas: static methods, one per number to compute.

```
    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;
    }
```

```
1/2 + 1/3 = 5/6
1/3 + 3/4 = 13/12
1/2 * 2/3 = 2/6
1/4 * 2/3 = 2/12
```

Listing 4.1*: static methods, since nothing is remembered yet; Problem 4.6 cures the loose numbers. Book: Sections 4.3.3–4.3.6 and Listings 4.1–4.3.*

4.1: A Tempting Wrong Solution

Listing 4.2 `WrongFractionSum.java`: three tempting ways to add $\frac{1}{2}$ and $\frac{1}{3}$.

```java
/** 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
```

CHECK YOURSELF. `printDifference(1, 3, 3, 4)`? — $\frac{e_1 d_2 - e_2 d_1}{d_1 d_2}$: $4 - 9 = -5$ over 12, so `1/3 - 3/4 = -5/12`.

Listing 4.2*: adding tops and bottoms is simply false;* `1 / 2` *is* `0` *in* `int`*; the* `double` *is close but not what was asked. A debugger shows the second at once. Book: Section 4.3.7 and Listing 4.4.*

Problem 4.2 — Analog Clock Angle

PROBLEM 4.2. Given `hours` (0–23) and `minutes` (0–59), `Q12Time` computes the angle **counterclockwise from the hour hand to the minute hand**, rounded and normalized to 0–359. Test 9:00, 3:00, 18:00, 1:00, 2:30 and 4:41: 270, 90, 180, 30, 255 and 254.

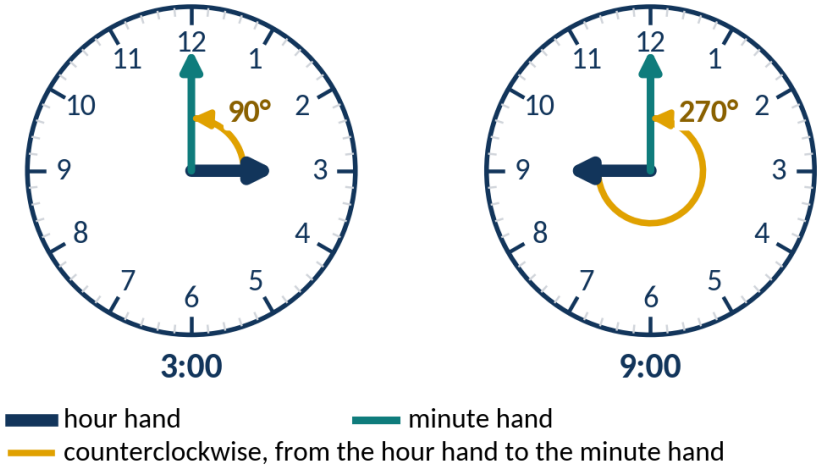


Figure 4.2 The clock at 3:00 (left) and at 9:00 (right); the amber arc runs counterclockwise from the hour hand (short, dark) to the minute hand (long, teal), giving 90° and 270°.

***Figure 4.2**: at 3:00 the angle is* 90°, *at 9:00* 270°. *One minute is* 6°, *one hour* 30°. *Book: Section 4.4.1 and Figure 4.2.*

4.2: Understanding It by Hand

$$\theta_{\text{hour}} = 30\left(h \bmod 12 + \frac{m}{60}\right) \tag{4.3}$$

$$\theta_{\text{minute}} = 6m \tag{4.4}$$

$$\alpha = \text{round}((\theta_{\text{hour}} - \theta_{\text{minute}}) \bmod 360) \bmod 360 \tag{4.5}$$

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 hour hand creeps (Equation (4.3)); counterclockwise means hour minus minute; normalize, then round (Equation (4.4), Equation (4.5)). ***Table 4.2****: at 4:41 the exact angle is* 254.5° *and the answer is 254 —* ***round half to even****. Book: Section 4.4.2 and Table 4.2.*

4.2: Normalize, Then Round

Listing 4.3 `Q12Time.java`, the method `angle`: Java needs `+ 360` and `Math.rint`.

```java
    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
    }
```

Listing 4.4 `q12_time.py`, the function `angle`: Python's `%` and `round` already behave as required.

```python
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
```

```
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.
```

*In Java `-105.0 % 360` is `-105.0`, so **Listing 4.3** adds 360; in Python `%` with a positive divisor is never negative (**Listing 4.4**). `minutes / 60.0` avoids integer division; at 2:11,* 359.5 *rounds to 360, and `% 360` makes it 0. Book: Sections 4.4.4–4.4.6 and Listings 4.6–4.8.*

4.2: Three Slips, and One Lucky Answer

```
difference:         -105.5
no +360:            -105
Math.round:         255
minutes / 60 int:   234
difference:         -105.5
round, then %:      254
round half up:      255
minutes // 60:      234
```

Java: no `+ 360` gives -105; `Math.round` rounds -105.5 *up* and gives 255; `minutes / 60` freezes the hour hand: 234.

Python's `round(-105.5)` is -106 (even), and $-106 \bmod 360 = 254$ — right, but *by luck*: it rounds before normalizing.

> **CHECK YOURSELF.** Clockwise at 4:41? $246 - 140.5 = 105.5$, half to even: 106. At 3:00: 270. (Check: $254.5 + 105.5 = 360$.)

A program that is right by accident is wrong after the next small change. Book: Section 4.4.7 and Listings 4.9–4.10.

Problem 4.3 — Computer Account

PROBLEM 4.3. `Q13ComputerAccount` is built from `realName`, `userName` and `password`, with `printRealName`, `printUserName`, `printPassword` and `changePassword(String)`; test it in `Q13ComputerAccountTest`.

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 first object that remembers. The last row of **Table 4.3** is the whole point of objects: one class, many independent objects. Book: Sections 4.5.1–4.5.2 and Table 4.3.*

4.3: Private Fields, a Controlled Check

Listing 4.5 `Q13ComputerAccount.java`, part 1: three private fields and the constructor.

```
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;
    }
```

Listing 4.6 `Q13ComputerAccount.java`, part 2: `hasPassword`, a controlled way to check the password.

```
    // Answers "is this the password?" without ever handing the password out.
    public boolean hasPassword(String attempt) {
        return password.equals(attempt);
    }
```

*In **Listing 4.5**, forget `this.` and the field stays `null`. There is deliberately no `getPassword`: **Listing 4.6** answers with `equals`. Book: Sections 4.5.3–4.5.6 and Listings 4.11–4.13.*

4.3: `==` Is Not `equals`

Listing 4.7 `WrongPasswordCheck.java`: `==` compares object identity, `equals` compares characters.

```java
/** 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
```

> **CHECK YOURSELF.** `changePassword(old, new)` returning `boolean`: `("wrong", "x")` returns `false`; `("sesame", "x")` returns `true`.

*In **Listing 4.7** a user typing the right password is refused; `==` seems to work on literals, so the bug hides in quick tests. Python: `is` is identity, `==` content. Book: Section 4.5.7 and Listing 4.14.*

Problem 4.4 — Complex Numbers

PROBLEM 4.4. `Q14ComplexNumber` with a constructor and getters, and methods to add, subtract and multiply two complex numbers and to write one as `"a + bi"`, using Equation (4.6) for the product.

$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i \tag{4.6}$$

$c_1 = 1 + 2i$, $c_2 = 3 + 4i$: sum $4 + 6i$; difference $-2 - 2i$; product $1 \cdot 3 - 2 \cdot 4 = -5$ and $1 \cdot 4 + 2 \cdot 3 = 10$, so $-5 + 10i$. Edge: $i \cdot i = -1 + 0i$.

Design decision: `add` takes *another* complex number and returns a *new* one, like $c_1 + c_2$ in mathematics. We print `-2.0 - 2.0i`, not `-2.0 + -2.0i`.

Book: Sections 4.6.1–4.6.3 and Equation (4.6).

4.4: Operations That Return New Objects

Listing 4.8 `Q14ComplexNumber.java`, the three operations: each returns a new object.

```java
    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);
    }
```

Listing 4.9 `q14_complex_number.py`, `__str__`: a minus sign and the positive part for a negative imaginary part.

```python
    def __str__(self):
        if self._imaginary < 0:
            return f"{self._real} - {-self._imaginary}i"
        return f"{self._real} + {self._imaginary}i"
```

*In **Listing 4.8**, `other.real` is allowed: privacy belongs to the class. **Listing 4.9** matches Java's `toString`. Book: Sections 4.6.4–4.6.6 and Listings 4.16–4.18.*

4.4: A Tempting Wrong Solution

Listing 4.10 `WrongComplex.java`: multiplying part by part, and printing a negative part naively.

```java
/** 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
```

CHECK YOURSELF. `c1.multiply(c1.conjugate())` for $1 + 2i$? — `5.0 + 0.0i`, since $(a + bi)(a - bi) = a^2 + b^2$.

*Part by part in **Listing 4.10** forgets the cross terms and $i \cdot i = -1$. Book: Section 4.6.7 and Listing 4.19.*

Problem 4.5 — Weight Conversion

PROBLEM 4.5. `Q15Weight(double p)` creates a weight in pounds, with `getPounds` and `getKilograms`, using Equation (4.7); test it in `Q15WeightTest`.

$$1 \text{ pound} = 0.45359237 \text{ kilograms} \tag{4.7}$$

By hand: $150 \times 0.45359237 = 68.0388555$; $140 \times 0.45359237 = 63.5029318$; 11 st 6 lb $=160$ lb $=72.5747792$ kg.

Store both pounds and kilograms? No: a future method could change one and forget the other, and the object would contradict itself.

The factor in Equation (4.7) is exact by international agreement, so all eight digits are written. Book: Sections 4.7.1–4.7.3 and Equation (4.7).

4.5: Store Once, Derive the Rest

Listing 4.11 `Q15Weight.java`: one constant, one field, a constructor and two getters.

```java
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 4.11 *stores the pounds only and computes the kilograms at every call, which costs one multiplication.* `static final` *means one copy, never changed. Book: Sections 4.7.4–4.7.6 and Listings 4.21–4.23.*

4.5: Tolerance, and a Value That Goes Stale

```
John: 150.0 lb = 68.0388555 kg
Mary: 140.0 lb = 63.502931800000006 kg
11 st 6 lb: 160.0 lb = 72.57477920000001 kg
```

Listing 4.12 `WrongWeight.java`, the setter: it forgets the second field.

```
    public void setPounds(double p) {
        pounds = p; // forgets to update kilograms
    }
```

```
140.0 lb = 68.0388555 kg ?
```

*`63.502931800000006` is not wrong: compare with `Math.abs(expected - actual) < 1e-9` (Python `assertAlmostEqual`). **Listing 4.12** claims 140 lb are 68.04 kg. Store each fact once, and compute everything that follows from it. Book: Sections 4.7.4–4.7.7 and Listing 4.24.*

Problem 4.6 — A Fraction Class

PROBLEM 4.6. `Q16Fraction` with `getNumerator`, `getDenominator`, `toString`, and `getSum` and `getProduct` returning a `Q16Fraction`. With $f_1 = \frac{1}{2}$ and $f_2 = \frac{3}{7}$: `f1.toString()` is `"1/2"`, `f2.getProduct(f1)` prints `3/14`, `f2.getSum(f1)` prints `13/14`.

By hand: $\frac{3 \cdot 1}{7 \cdot 2} = \frac{3}{14}$; $\frac{3 \cdot 2 + 7 \cdot 1}{14} = \frac{13}{14}$; and afterwards f_2 must still be $\frac{3}{7}$.

Chaining: $(f_1 + f_1) \cdot (f_2 + f_2) = \frac{4}{4} \cdot \frac{42}{49} = \frac{168}{196}$, which is just $\frac{6}{7}$.

Problem 4.1 again, but each fraction is now one object. Book: Sections 4.8.1–4.8.2.

4.6: New Object, or Changed Object?

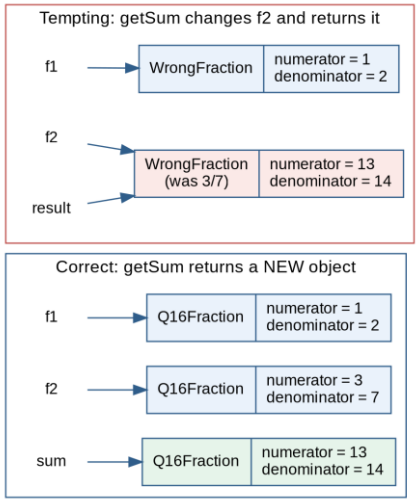


Figure 4.3 Lower box: the correct `getSum` leaves `f1` and `f2` alone and returns a new object. Upper box: the tempting version of Section 4.8.7 overwrites `f2` and returns it, so `f2` and the result are the same changed object.

***Figure 4.3**, lower box: the correct `getSum` returns a third object. An object that never changes is immutable — like Java's `String`. Book: Section 4.8.3 and Figure 4.4.*

4.6: An Immutable Fraction

Listing 4.13 `Q16Fraction.java`, the two operations: each builds and returns a new fraction.

```
    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 4.14 `q16_fraction.py`, the same two operations in Python.

```
    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())
```

*No setters makes **Listing 4.13** immutable; **Listing 4.14** translates it line for line. The driver prints `1/2`, `3/14`, `13/14`. Book: Sections 4.8.4–4.8.6 and Listings 4.26–4.28.*

4.6: A `getSum` That Changes `this`

Listing 4.15 `WrongFraction.java`, the tempting `getSum`: it overwrites its own fields and returns `this`.

```
    public WrongFraction getSum(WrongFraction other) {
        numerator = numerator * other.denominator + denominator * other.numerator;
        denominator = denominator * other.denominator;
        return this;
    }
```

```
f2 + f1 = 13/14
f2 is now 13/14
f2 + f1 = 40/28 (same call, new answer)
```

> **CHECK YOURSELF.** Reduce in the constructor with Euclid's gcd: `new Q16Fraction(2, 6)` prints `1/3`, and `f2.getSum(f2)` prints `6/7`.

*The same call, `f2.getSum(f1)`, gives a new answer the second time: the upper box of **Figure 4.3**. **Listing 4.15** is a trap for everyone who uses it. Book: Section 4.8.7 and Listing 4.29.*

Problem 4.7 — Player and Enemy

PROBLEM 4.7. `Player` and `Enemy`, each with name, health, power and defense, constructor, getters and setters; `attack(opponent)` makes the opponent call `takeDamage(attack)`, which reduces health by (attack − defense) and prints `"{name} died!"` below zero.

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!"

*Each object is responsible for its own fields. The statement is silent on negative damage (we treat it as zero) and slightly off at exactly 0 health (we say dead): write such decisions down (**Table 4.4**). Book: Sections 4.9.1–4.9.3 and Table 4.4.*

4.7: One Message, and the Defender Decides

Listing 4.16 `Player.java`, the fight: `attack` sends one message; `takeDamage` works out the damage from its own 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!");
        }
    }
```

```
Round 1: Villain has 40 health left
Round 2: Villain has 20 health left
Villain died!
Round 3: Villain has 0 health left
```

Subtract first, then test. `Villain died!` *appears inside the third attack, before* `Round 3` *is printed: output order records call order.* `Enemy.java` *is the mirror image of* ***Listing 4.16****; Chapter 7 lets them share code. Book: Sections 4.9.4–4.9.6 and Listings 4.31–4.33.*

4.7: Reaching into Another Object

Listing 4.17 `Cheater.java`: code outside `Enemy` tries to write its private field.

```java
/** 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());
    }
}
```

```
Cheater.java:5: error: health has private access in Enemy
        villain.health = 0;
               ^
1 error
```

> **CHECK YOURSELF.** Add `isAlive()` and let a dead character ignore attacks: one more `hero.attack(villain)` prints nothing and health stays 0.

*The compiler error for **Listing 4.17** is a good thing. Python's `villain._health = 0` goes through silently — and no "died!" is printed, because the one method that knows about death was bypassed. Book: Section 4.9.7 and Listings 4.34–4.35.*

Problem 4.8 — Money in Rupiah

PROBLEM 4.8. `Rupiah` holds whole rupiah, with `getAmount`, and `plus`, `times` and `discount(percent)` (rounded down) returning new objects; `toString` groups digits from the right: `Rp1.250.000`, `Rp999`, `-Rp5.000`.

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`

Money is a `long`, never a `double`. $4 \cdot 7000 + 12500 = 40500$*; 10% off is* $\frac{40500 \cdot 90}{100} = 36450$*; multiply before dividing.* ***Table 4.5*** *groups 1250000. Book: Sections 4.10.1–4.10.2 and Table 4.5.*

4.8: Grouping from the Right

Listing 4.18 `Rupiah.java`, `toString`: the procedure of the hand trace written as a loop that counts down.

```
    @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;
    }
```

```
Order: Rp40.500
After 10% off: Rp36.450
```

*`count % 3 == 0 && i > 0`: without `i > 0`, `999` would become `.999`. Python walks `range(len(digits) - 1, -1, -1)` the same way. **Listing 4.18** is immutable like `Q16Fraction`. Book: Sections 4.10.3–4.10.6 and Listings 4.36–4.38.*

4.8: From the Left, and in a `double`

Listing 4.19 `wrong_rupiah_format.py`: the same two mistakes in Python.

```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.** Rp100.000 split among 3: Rp33.333 each and Rp1 left over — without `remainder`, one rupiah would silently vanish.

Groups of three belong to thousands and millions, counted from the right; 1.1 *has no exact binary form.* ***Listing 4.19*** *shows both mistakes; the Java version prints the same. Book: Section 4.10.7 and Listing 4.40.*

Problem 4.9 — Time of Day

PROBLEM 4.9. `TimeOfDay(hours, minutes)` on a 24-hour clock, with getters, `plusMinutes(m)` wrapping around midnight in both directions, `isBefore`, `minutesUntil`, and `toString` as `08:05`.

22:30 + 45 min = 23:15 — minutes *carry*. 22:30 + 150 min = 01:00 — hours *wrap*. 00:10 − 20 min = 23:50.

One idea: store one number, minutes since midnight (0–1439). $22 \cdot 60 + 30 = 1350$; $+150 = 1500$; $1500 \bmod 1440 = 60$, which is 01:00. And -10 wraps to 1430: 23:50.

The outside speaks hours and minutes; the inside uses the representation that makes the arithmetic easy. Choosing it, and hiding it, is what encapsulation buys. Book: Sections 4.11.1–4.11.3.

4.9: One Internal Number, Hidden

Listing 4.20 `TimeOfDay.java`, the core: the constructor, a private `wrap`, and arithmetic on minutes.

```
    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);
    }
```

Listing 4.20 *prints* `Dropped off at 22:30, ready at 01:00`*; Python's* `%` *needs no* `+ 1440`*. Book: Sections 4.11.4–4.11.6 and Listings 4.41–4.43.*

4.9: No Carry, No Wrap, No Padding

Listing 4.21 `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
```

> **CHECK YOURSELF.** Open 07:00–21:00: 06:59 closed, 07:00 open, 20:59 open, 21:00 closed — include the opening minute, exclude the closing one.

*Each line of **Listing 4.21** is a real bug from real booking systems; all three vanish with one number kept in range. Book: Section 4.11.7 and Listing 4.44.*

Problem 4.10 — A Laundry Receipt

PROBLEM 4.10. `LaundryReceipt(customer, express)`; `addService(service, kg)` adds a line — wash Rp7.000, iron Rp5.000, wash+iron Rp10.000 per *started* kilogram, express +50% — or returns `false` for an unknown service or a weight that is not positive. The total is a `Rupiah`.

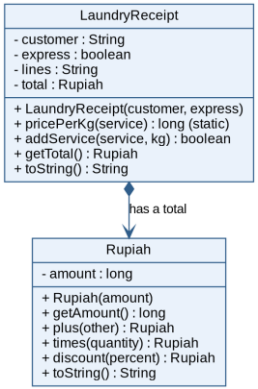


Figure 4.4 The class diagram of Problem 4.10: a `LaundryReceipt` has a `Rupiah` total; the filled diamond marks that the total belongs to the receipt.

Sari: $4 \cdot 7000 + 2 \cdot 5000 = 38000$*; Budi:* $50000 + 25000 = 75000$. ***Figure 4.4****: a receipt has a* `Rupiah` *—* ***composition****. Book: Sections 4.12.1–4.12.3 and Figure 4.5.*

4.10: Refuse Early, Charge by the Ceiling

Listing 4.22 `LaundryReceipt.java`, `addService`: refuse early, charge per started kilogram, total in `Rupiah`.

```
    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);
        // 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;
    }
```

*`pricePerKg` is a `switch` on a string, which compares with `equals`. `Locale.ROOT` keeps `3.2` from becoming `3,2` on an Indonesian computer. **Listing 4.22** and its Python twin print the same receipts. Book: Sections 4.12.4–4.12.6 and Listings 4.46–4.48.*

4.10: The Receipts, and Three Ways to Get 3.2 Wrong

```
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
int(kg):       3 kg -> Rp21000
round(kg):     3 kg
round(2.5):    2 kg
math.ceil(kg): 4 kg
```

> **CHECK YOURSELF.** A minimum charge of Rp20.000 belongs in `getTotal()`; 1.5 kg of ironing costs Rp10.000, so the total is Rp20.000.

`int(3.2)` loses Rp7.000 on one bag; `round` gives 3 too, and `round(2.5)` is 2 (half to even). "Per started kilogram" means the ceiling. Book: Section 4.12.7 and Listings 4.49–4.50.

Problem 4.11 — Tidying a Name

PROBLEM 4.11. `PersonName` keeps the raw text `"  siti  NUR  aisyah  "` and offers `wordCount()`, `initials()` (`"S.N.A."`) and `tidy()` (`"Siti Nur Aisyah"`), using only `length`, `charAt` and character methods.

```
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?          ^              ^           ^
```

A character starts a word if it is not a space and is first or follows a space: positions 2, 9 and 13. Edge cases: the empty name, a name of three spaces. Book: Sections 4.13.1–4.13.2.

4.11: One Private Rule, Used Three Times

Listing 4.23 `PersonName.java`, the rule: where a word starts.

```
    private boolean startsWord(int i) {
        return raw.charAt(i) != ' ' && (i == 0 || raw.charAt(i - 1) == ' ');
    }
```

Listing 4.24 `PersonName.java`, `tidy`: capitalize at word starts, lower-case elsewhere, drop spaces.

```
        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 4.23 *relies on* ***short-circuit*** `||`*;* ***Listing 4.24*** *adds a space before every word but the first. Book: Sections 4.13.3–4.13.6 and Listings 4.51–4.53.*

4.11: `split(" ")` and Empty Words

Listing 4.25 `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
```

*In **Listing 4.25** every extra space makes an empty word `''`, and `''[0]` fails; Java stops with `StringIndexOutOfBoundsException`. Shortcuts: Python `split()`, Java `trim().split("\\s+")`. Check yourself: `lastNameFirst()` gives `"Aisyah, Siti Nur"`. Book: Section 4.13.7 and Listing 4.55.*

Problem 4.12 — Password Rules

PROBLEM 4.12. `PasswordChecker.problems(password)` lists every broken rule, separated by `"; "`: at least 8 characters, a digit, a capital letter, a small letter; `isStrong` is true exactly when there are none.

`sesame` → `too short; no digit; no capital letter`. `1sesame` → `too short; no capital letter`. `Sesame2026` → strong.

Boundary: `Abcdefg1` (8) strong, `Abcdef1` (7) too short.

"Some character is a digit" can be decided at the first digit; "no character is a capital" only after the *last* character.

Modern guidance (NIST SP 800-63B) prefers long passwords checked against common ones, and a real system stores only a salted hash. Book: Sections 4.14.1–4.14.3.

4.12: Count in the Loop, Decide After It

Listing 4.26 `PasswordChecker.java`, `problems`: one loop counts, `if` statements after it decide.

```
        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");
        }
```

*In **Listing 4.26**, an `else if` chain because a character is at most one kind; the `add` helper puts `"; "` between problems; `isStrong` is defined through `problems`, so they never disagree. Book: Sections 4.14.4–4.14.6 and Listings 4.56–4.58.*

4.12: A Verdict After One Character

Listing 4.27 `WrongPassword.java`, `hasDigit`: the `else` returns inside the loop.

```
    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;
    }
```

```
1sesame:  true
Laundry7: false
```

CHECK YOURSELF. A fifth rule, `"no symbol"`: `Sesame2026` now reports `no symbol`; `Sesame-2026` is strong.

Listing 4.27 *looks only at the first character — not really a loop at all; step through* ***`hasDigit("Laundry7")`*** *and watch* ***`i`*** *never reach 1. Move* ***`return false`*** *after the loop. Book: Section 4.14.7 and Listing 4.59.*

Problem 4.13 — Tracing Loops by Hand

PROBLEM 4.13. Without running a program, trace (a) the digit sum of $n = 4071$ and (b) Euclid's algorithm for $a = 84$, $b = 36$; then write a program whose trace reproduces your tables.

Table 4.6 Trace of the digit-sum loop for $n = 4071$: four passes, one per digit, ending when n reaches 0.

Step	digit $= n \bmod 10$	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

Table 4.6*: four passes, one per digit;* $4 + 0 + 7 + 1 = 12$*. For* $n = 0$ *the loop runs zero times. A trace table is the paper version of the debugger's watch window. Book: Sections 4.15.1–4.15.2 and Table 4.6.*

4.13: Euclid's Algorithm by Hand

$$\gcd(a, b) = \gcd(b,\ a \bmod b), \qquad \gcd(a, 0) = a \tag{4.8}$$

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	$r = a \bmod b$	New a, new b
84	36	$84 - 2 \cdot 36 = 12$	36, 12
36	12	$36 - 3 \cdot 12 = 0$	12, 0

$\gcd(84,36) = 12$: $84 = 7 \cdot 12$, $36 = 3 \cdot 12$. Starting with $\gcd(36,84)$ just swaps the numbers in the first pass.

*Equation (4.8): anything dividing a and b divides a mod b; **Table 4.7** applies it twice. Book: Section 4.15.2, Equation (4.8) and Table 4.7.*

4.13: Loops That Print Their Own Trace

```
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
```

`n = n / 10` is integer division in Java; Python *must* write `n // 10`. With `/`, Python's loop runs 328 passes and sums fractions to `13.33...`.

`while (n > 9)` feels right and stops one pass early: 8 instead of 12 — the table would have three rows instead of four.

*The printed rows are **Table 4.6** and **Table 4.7**. Check yourself: gcd(1071,462) takes three passes and is 21. Book: Sections 4.15.4–4.15.7 and Listings 4.61–4.65.*

Problem 4.14 — Leap Years and the Next Day

PROBLEM 4.14. `SimpleDate` with static `isLeapYear` and `daysInMonth`, `isValid()`, `nextDay()` returning a new date, and `toString` as `2024-02-29`; also count the leap years from 1901 to 2100.

$$\text{leap}(y) = (4 \mid y \ \wedge \ 100 \nmid y) \ \vee \ 400 \mid y \tag{4.9}$$

2024 leap, 2023 not, 1900 not ($100 \mid 1900$, $400 \nmid 1900$), 2000 leap. From 1901 to 2100: 50 multiples of 4, minus 2100: **49**.

`nextDay`: not the last day → day + 1; else not December → 1st of next month; else 1 January next year.

Equation (4.9) is the Gregorian rule as one Boolean expression. Book: Sections 4.16.1–4.16.3 and Equation (4.9).

4.14: A Rule as One Boolean Expression

Listing 4.28 `SimpleDate.java`, the calendar facts: leap years, and month lengths by `switch`.

```
    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;
        }
    }
```

*The facts are static: they belong to the calendar, not to one date (**Listing 4.28**); several `case` labels share one body. Book: Sections 4.16.4–4.16.6 and Listings 4.66–4.68.*

4.14: The Next Day, and the Julian Trap

Listing 4.29 `simple_date.py`, `next_day`: three cases, checked in order, each returning a new date.

```
    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)
```

```
1900 leap? true
Leap years from 1901 to 2100: 50
```

CHECK YOURSELF. `dayOfYear()` with a `for` loop over earlier months: 2024-03-01 is day 61; 2023-12-31 is day 365.

Listing 4.29 *gives 2024-02-29 after 2024-02-28. "Divisible by 4" alone is the Julian rule, right for 1901–2099 and wrong for 1900 and 2100, as the output shows. Book: Section 4.16.7 and Listing 4.69.*

The Seven Classic Problems Side by Side

Table 4.8 Problems 4.1–4.7: 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

Table 4.8*: every class that represents a value was made immutable. Book: Section 4.17 and Table 4.8.*

The Seven New Problems Side by Side

Table 4.9 Problems 4.8–4.14: the design idea each one teaches, and the trap its tempting wrong solution demonstrates.

Problem	Design idea	Trap shown by the tempting wrong solution
4.8 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

***Table 4.9**: almost every trap is invisible on the “nice” examples and visible only at an edge. Testing the edges is testing. Book: Section 4.17 and Table 4.8.*

Profitina in Practice

Small classes, each tested on its own: each signal of the scanner is one small class; well over a thousand tests must be green before a release — PASS/FAIL at full size.

Money and exact numbers: the PROFIT coin, with six decimals, is handled as whole smallest units, and shown to Indonesian users grouped with dots.

Passwords are never printed: stored only as an Argon2id hash, changed only with proof of the old one (UU PDP 27/2022).

Time and dates: weekly score trends and scheduled re-scans meet the traps of Problems 4.9 and 4.14.

A score tells a brand owner where to look; the owner decides what to change. Book: Section 4.18.

Common Mistakes

Dividing two integers and expecting a fraction · trusting `%` with negatives in Java · `==` on strings · rounding with the wrong rule.

Comparing `double` values with `==` · storing the same fact twice · changing `this` in a `getSum` · reaching into another object.

Deciding inside a loop what only the end can decide · off-by-one conditions · forgetting `this.` in a constructor.

Testing only the examples in the statement. Add zero, negatives, boundaries, an empty string, an odd minute, a century year.

Every tempting wrong solution today passed at least one “nice” example. Book: Section 4.19.

Summary — Five Things to Remember

1. Eight steps: restate, examples by hand, nouns and verbs, class box, tests first, small pieces, run and debug, look back.

2. Plain Java drivers with `check` and Python's `unittest` turn hand examples into automatic checks; they move to JUnit 6 mechanically.

3. Value classes — fractions, complex numbers, money, times, dates — are immutable; their operations return new objects.

4. Store each fact once; choose a good internal representation and hide it; let each object guard its own fields.

5. Know your arithmetic: integer division, Java's `%`, half-to-even (Equation (4.5)), the ceiling, tolerances; count in a loop, decide after it.

Book: Section 4.20.

Next Time

Chapter 5 — Arrays, Lists, Input, Exceptions, and Files.

A receipt that holds any number of lines as objects; keyboard input; exceptions that can be caught, like the `StringIndexOutOfBoundsException` of Problem 4.11; files.

Before we meet: Exercises 4.2, 4.5, 4.8 and 4.12 — clock angles, a traced `toString`, a traced digit sum, and the eight steps on your own. Solutions are in Section 4.22.

Book: Sections 4.21–4.22, then Chapter 5.

Lecture 5 — Arrays, Lists, Input, Exceptions, and Files

Course: Object-Oriented Programming (EF234302)

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*, Chapter 5

Lecturer: Irfan Subakti — yifana@gmail.com

Today our programs stop assuming that they know how many things they will handle, and that nothing will go wrong. Book: Chapter 5, Sections 5.1–5.12.

What This Meeting Covers

1. **Arrays** — a fixed row of numbered slots; index 0 to $n - 1$; references, copies and `args`.
2. **Lists** — `ArrayList` and Python's `list`, the everyday operations, how a list grows, boxing.
3. **Input** — `Scanner` and `input()`, the leftover line break, sentinel loops.
4. **Exceptions** — stack traces, `try`/`catch`/`finally`, `throw`, checked versus unchecked, exceptions of our own.
5. **Files** — streams and readers, closing what we open, writing text, CSV.
6. **Profitina Laundry** — loading orders from a damaged CSV file, rejecting each bad line precisely, saving a clean file.

The order follows the order in which the ideas depend on each other: arrays, lists, input, exceptions, files. Book: Section 5.1.

Where We Left Off: Three Gaps

Chapter 3 gave us classes, objects, strings, types, conditionals and loops; Chapter 4 worked problems with all of them. Every variable held **one** value.

Many values. On a Monday the laundry takes four orders or forty; we cannot declare forty variables.

Input. A program that only prints is a speech, not a conversation.

Failure. A customer types “three” for 3; a file has been moved or edited by hand. A program that meets these with a wall of red text is a prototype, not a finished program.

Exceptions and files belong together: nearly everything that touches a file can fail, and Java will not even compile file code until we say what happens when it does. Book: Section 5.1.

The Array: a Pigeonhole Rack

> **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 $0, 1, \ldots, n-1$.

The index is a **distance** from the start: if the array begins at address b and each slot is s bytes wide, slot i begins at the address of Equation (5.1).

$$\text{address}(i) = b + i \cdot s \tag{5.1}$$

One multiplication and one addition reach any slot, however long the array: *random access*.

> **REMEMBER.** An array of length n has **no** element with index n.

We have met an array already: the `String[] args` of every `main`. Book: Section 5.2.1.

An Array in Memory

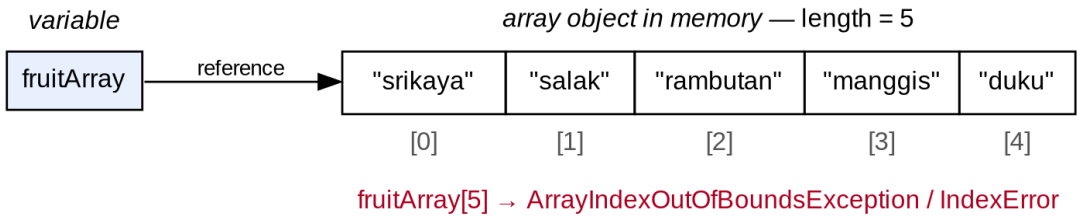


Figure 5.1 An array of five strings in memory. The variable holds a reference to the array object; the slots are numbered from 0 to 4, and index 5 does not exist.

*In **Figure 5.1** the variable `fruitArray` does not hold the five fruits; it holds a reference to the array object, which also records its length. Book: Section 5.2.1.*

Creating and Filling an Array

Listing 5.1 builds five `String` slots with `new String[5]` and fills them one index at a time.

Listing 5.1 From `main` in `FruitArray.java`: `new`, default values, assignment by index, and the counting loop.

```
        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]);
        }
```

A new array is never empty: every slot holds a default value, `null` for objects, `0` for `int`, `false` for `boolean`. Always write `i < fruitArray.length`, never a hard-coded 5. Book: Section 5.2.2.

Shorthand and the Enhanced `for`

Listing 5.2 is the rest of `main`: the braces form counts the values for us, and `for (String fruit : …)` visits each value without an index.

Listing 5.2 The rest of `main` in `FruitArray.java`: the shorthand initializer, the enhanced `for`, and an array of `int`.

```
        // 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));
```

```
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]
```

`length` is a field of the array, with no parentheses; strings use the method `length()`. `Arrays.toString` turns an array into readable text. Book: Section 5.2.2.

The Same Rack in Python

Python has no fixed-size array in the language; **Listing 5.3** imitates one with a list of five `None`s.

Listing 5.3 The first half of `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])
```

`len(x)` plays the part of `.length`; `range(n)` gives $0, 1, \ldots, n-1$*; Python does not fill a list with zeros unless we say `[0] * 10`. The output matches the Java output line for line. Book: Section 5.2.2.*

Stepping Off the End

Listing 5.4 writes the classic off-by-one loop: `<=` where `<` belongs.

Listing 5.4 `ArrayBounds.java`: a loop that runs one step too far and writes past the end of a five-element array.

```java
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 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)
```

Java throws, names the line (line 8), and abandons the rest of `main`. Python raises `IndexError` in the same place; and since `a[-1]` is the last element in Python, a wrongly computed −1 fails silently there. Book: Section 5.2.3.

Arrays of Arrays

Listing 5.5 totals the loads of three machines over three days with an `int[][]`: `washes[m][d]` is row `m`, column `d`.

Listing 5.5 From `LoadStats.java`: row sums of a two-dimensional array.

```
        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");
        }
```

```
Machine 1 washed 36 loads
Machine 2 washed 33 loads
Machine 3 washed 33 loads
Machine 2 on day 3: 14
```

The first half of the program keeps a running total and a running maximum that starts at the first element, not at 0; average $27.0/5 = 5.40$*. "Machine 2 on day 3" is* `washes[1][2]` *because both indexes start at 0. Book: Section 5.2.4.*

An Array Variable Holds a Reference

Listing 5.6 shows that assignment copies the **arrow**, not the array, and how to make a real copy.

Listing 5.6 From `ArrayCopy.java`: aliasing, `Arrays.copyOf`, and comparing with `==` and `Arrays.equals`.

```
        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[] twin = {99, 2, 3};
        System.out.println("original == twin? " + (original == twin));
        System.out.println("Arrays.equals(original, twin)? " + Arrays.equals(original, twin));
```

`original` becomes `[99, 2, 3]` through its alias; `==` is `false`, `Arrays.equals` is `true`. An array cannot grow: `Arrays.copyOf(original, 6)` makes a bigger one and copies. In Python, `is` asks "same object?" and `==` compares contents. Book: Section 5.2.5.

The Array in `main`: `args`

`java Hello2 one two three` puts three strings into `args` (**Listing 5.7**); Python puts them into `sys.argv`, after the script's own name (**Listing 5.8**).

Listing 5.7 `Hello2.java`: printing the command-line arguments.

```java
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]);
        }
    }
}
```

Listing 5.8 `hello2.py`: printing the command-line arguments from `sys.argv`.

```python
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])
```

Command-line arguments carry what varies from run to run: a file name, a date, a mode. In VS Code, add an `"args"` entry to `launch.json`. Book: Section 5.2.6.

Lists That Grow

The laundry does not know on Monday morning how many orders will arrive. Any fixed array size is wrong: too small and we crash, too large and we waste memory and must count the slots in use.

> **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.

`ArrayList<String>` reads "an `ArrayList` that holds `String`s, and only `String`s". Chapter 9 explains these *generic* types in full.

Java provides the growable list as a library class, `java.util.ArrayList`; Python builds it into the language. Book: Section 5.3.1.

The Everyday `ArrayList` Operations

Listing 5.9 adds six fruits, reads one, and asks where two of them are.

Listing 5.9 From `FruitList.java`: `add`, `size`, `get` and `indexOf`.

```
        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));
        int positionOfPeach = fruit.indexOf("peach");
        int positionOfKiwi = fruit.indexOf("kiwi");
        System.out.println("peach at " + positionOfPeach + ", kiwi at " + positionOfKiwi);
```

```
List: [mango, banana, peach, plum, durian, plum], size 6
First fruit: mango
peach at 2, kiwi at -1
```

`indexOf` returns −1 for "absent", which can never be a valid index. `contains` answers the yes-or-no question. Three spellings of "how many": `array.length`, `string.length()`, `list.size()`. Book: Section 5.3.2.

Replacing and the Two `remove`s

Listing 5.10: `remove(3)` removes the element at *position* 3 and returns it; `remove("plum")` removes the first equal element and returns `true` or `false`.

Listing 5.10 From `FruitList.java`: `set`, and removing by position and by value.

```
        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"));
```

```
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
```

*Later elements shift left, so indexes change. Python's `index` and `remove` **raise** `ValueError` for an absent value instead of returning −1 or `false`; removing by position is `pop(3)` or `del fruit[3]`. Book: Section 5.3.2.*

Array, `ArrayList`, `list`: Reading and Writing

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`

Table 5.1 *is the first half of the side-by-side comparison; keep it nearby for the rest of the course. Book: Section 5.3.2.*

Array, `ArrayList`, `list`: Growing and Searching

Table 5.2 The everyday operations on a Java array, a Java `ArrayList`, and a Python `list`.

Operation	Java array `a`	Java `ArrayList<T> list`	Python `list`
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`

Table 5.2 *is the second half: an array cannot append, insert or remove, and the last row names the exception each container throws for a bad index. Book: Section 5.3.2.*

How a List Grows

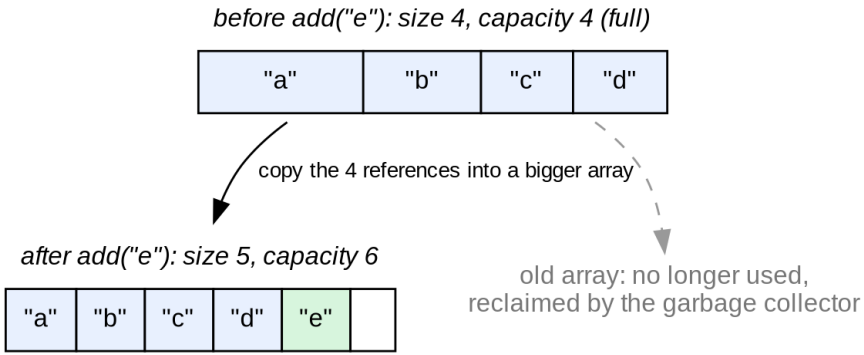


Figure 5.2 What `add` does when the hidden array is full: a larger array is made, the old references are copied into it, and the old array is thrown away.

*Inside every `ArrayList` is an ordinary array with spare slots; the list keeps its size and its capacity. When the array is full, `add` does what **Figure 5.2** shows. Book: Section 5.3.3.*

Grow by One, or Double?

Grow by one slot: adding n elements copies the total of Equation (5.2); for $n = 1000$ that is 499,500 copies.

$$0 + 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} \tag{5.2}$$

Double when full: copies happen at sizes 1,2,4, … , 2^k, and Equation (5.3) bounds the total; for $n = 1000$ it is 1023.

$$1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n \tag{5.3}$$

Fewer than two copies per element on average: appending is, on average, as cheap as writing into an array.

Java grows by a factor of 1.5*, CPython by a somewhat smaller factor; any factor greater than 1 gives the same argument with a different constant. Exercise 5.4 measures both strategies. Book: Section 5.3.3.*

Lists of Numbers: Boxing

`ArrayList<int>` does not compile: the type parameter must be a class, so we write `ArrayList<Integer>`. **Listing 5.11** shows the one famous trap.

Listing 5.11 From `BoxingDemo.java`: boxing, and the two meanings of `remove` on a list of `Integer`.

```
        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);
```

Output: `[5, 10, 3, 7]`, then `[5, 10, 3]`, then `[5, 10]`. `remove(3)` passes an `int`, which matches `remove(int index)` exactly. Python avoids the ambiguity with two words: `del weights[3]` for a position, `weights.remove(3)` for a value. Arrays suit fixed sizes and primitives; when in doubt, choose `ArrayList`. Book: Sections 5.3.4 and 5.3.5.

Reading the Keyboard with `Scanner`

Listing 5.12 reads a line, an integer, a decimal number and another line.

Listing 5.12 The body of `SquareInput.java`: reading with `Scanner`.

```
        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 + "\".");
```

`nextInt()`, `nextDouble()` and `next()` read one token; `nextLine()` reads the rest of the line. For repeatable runs, redirect a file: `java SquareInput < SquareInput.input.txt`. Book: Section 5.4.1.

The Leftover Line Break

`nextDouble()` reads `3.5` and stops **right after it**, leaving the line break unread. A `nextLine()` straight after would return `""`, and the courier note would be read by nobody.

> **RULE.** After `nextInt()`, `nextDouble()` or `next()`, call `nextLine()` once before you want a whole line — line 14 of **Listing 5.12**.

Python's `input()` always reads a whole line and returns a `str`; we convert with `int(...)` or `float(...)`, as **Listing 5.13** does. No leftover, no trap.

Listing 5.13 The end of `square_input.py`: nothing to swallow, because `input()` reads whole lines.

```
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 + '".')
```

If the user types `twelve`, both `nextInt()` and `int()` refuse and the program crashes; Section 5.5 is devoted to doing better. Book: Section 5.4.1.

Reading Until a Sentinel

Listing 5.14 reads fruit names until `#stop#`: input, a `while` loop and a growing list together.

Listing 5.14 The body of `FruitCollector.java`: reading lines into an `ArrayList` until the sentinel `#stop#`.

```
        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?");
```

```
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?
```

The priming read happens before the `while`, so the test always examines a line just read; strings are compared with `equals`, never `==`. If the input ends without `#stop#`, `nextLine()` throws `NoSuchElementException`. Book: Section 5.4.2.

What Is an 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.

Met so far: `ArrayIndexOutOfBoundsException`, `IndexError`, `ValueError`. Coming soon: `InputMismatchException`, `NullPointerException`.

One of the problems with programming is users: we can only guess what might go wrong and soften its effects. Book: Section 5.5.1.

A Program Meets a User

Listing 5.15 reads integers until 99 and trusts the user completely. The input is `1`, `2`, `b`, `3`, `99`.

Listing 5.15 The body of `InputTest.java`: reading integers until 99, with no protection against bad input.

```
        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());
```

```
Enter some numbers!
Adding 1
Adding 2
Exception in thread "main" java.util.InputMismatchException
    at InputTest.main(InputTest.java:12)
```

The middle of the trace (four lines inside `Scanner`) is omitted here; the book prints all of it. Book: Section 5.5.1.

Reading a Stack Trace

What: the first line names the type, `java.util.InputMismatchException`.

Where: scan down for the first line that names *your* class: `at InputTest.main(InputTest.java:12)`, the second `nextInt()`, which read `b`.

> **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.

A Python **traceback** runs the other way, "most recent call last": read it from the bottom up. `ValueError: invalid literal for int() with base 10: 'b'` even quotes the bad text.

The library's own file names and line numbers (`Scanner.java:947`) may differ between Java versions; your own line numbers do not. Book: Section 5.5.2.

Propagation: Three Calls Deep

In **Listing 5.16**, `main` → `loadAll` → `readOrder` → `parseWeight`, which throws for the weight `-2`.

Listing 5.16 The two lowest methods of `PropagationDemo.java`: the exception is thrown three calls deep.

```
    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");
    }
```

`loadAll` calls `readOrder` for each of `"Sari,3.5"`, `"Budi,-2"`, `"Tono,4"`, and `main` calls `loadAll` inside a `try` that catches `IllegalArgumentException`. Book: Section 5.5.2.

The Exception’s Diary of the Journey

The stack trace printed by **Listing 5.16** lists every method that was abandoned, most recent first:

```
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)
```

`Tono,4` is never read, and “All lines loaded.” is never printed: each method was abandoned at the call.

Line 5 in `parseWeight`, called from line 12 in `readOrder`, from line 18 in `loadAll`, from line 26 in `main`. Python’s `traceback.print_exc` gives the same trail, most recent call last. Book: Section 5.5.2.

Propagation, Drawn

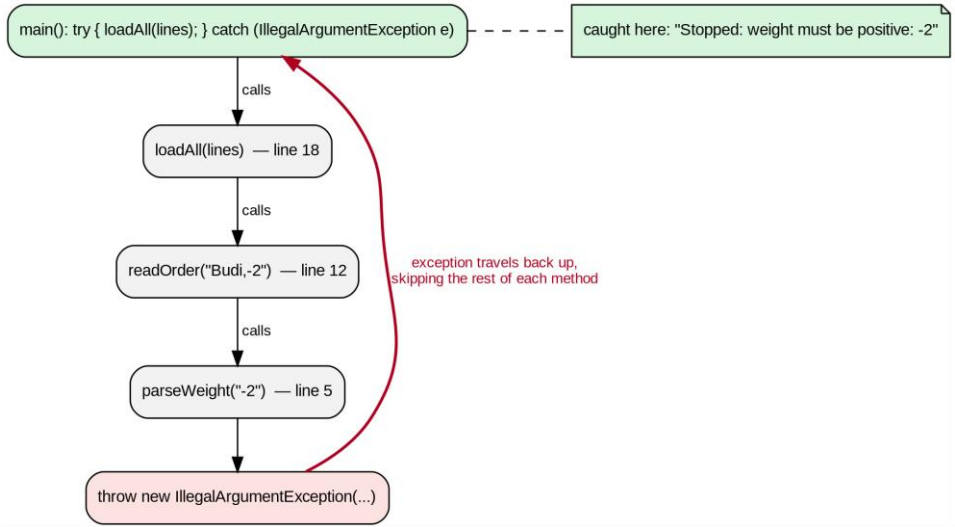


Figure 5.3 Propagation. The calls go down from `main`; the exception travels back up, abandoning each method on the way, until it reaches a matching `catch`.

***Figure 5.3**: this journey back up the chain of calls is propagation, and it stops at the first `catch` whose type matches. Book: Section 5.5.2.*

Java's Family Tree of Failures

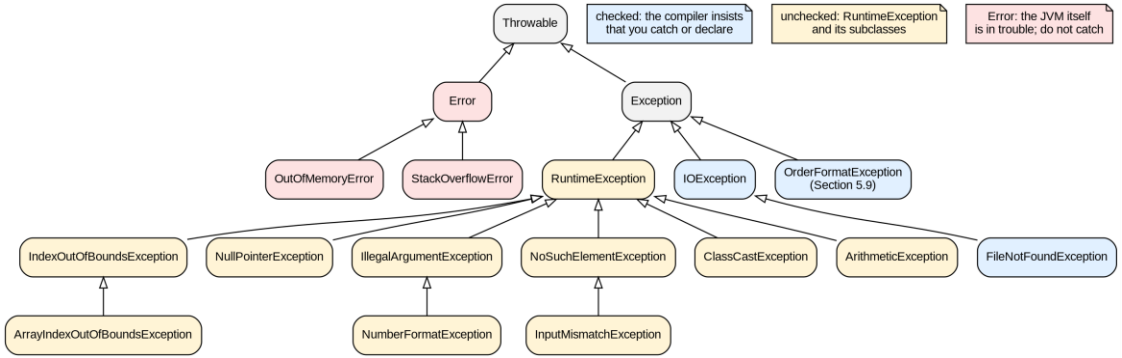


Figure 5.4 Part of Java's exception hierarchy. Blue classes are checked exceptions, yellow ones are unchecked, and red ones are errors that programs should not try to handle.

*In **Figure 5.4**, `Error` means Java cannot recover (out of memory, stack overflow): we do not catch it. Under `Exception`, the split between `RuntimeException` and the rest is the subject of Section 5.6. Book: Section 5.5.3.*

Python's Family Tree

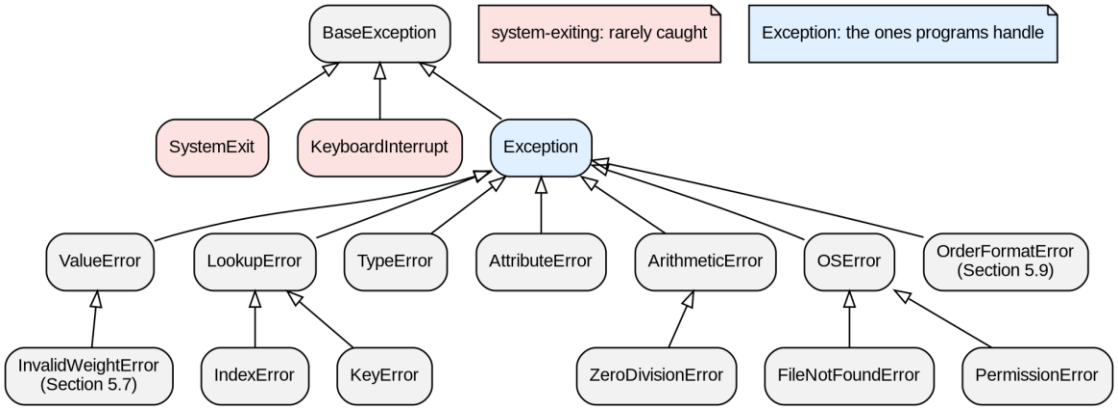


Figure 5.5 Part of Python's exception hierarchy. Programs catch subclasses of `Exception`; the exceptions beside it ask the program to stop.

***Figure 5.5**: a net labeled with a class also catches every descendant, so `except LookupError` catches both `IndexError` and `KeyError`. Book: Section 5.5.3.*

`try` and `catch`

No exception: the whole `try` block runs and the `catch` is skipped. An exception: the rest of the `try` is skipped; if the type matches (**or is a subclass**), the `catch` runs; if not, the exception keeps propagating.

Listing 5.17 repairs `InputTest` for the input `1 2 b 3 ten 99`:

Listing 5.17 The loop of `SafeInput.java`: catching `InputMismatchException` so that bad input is reported and skipped.

```
        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.");
            }
        }
```

Python writes `try:` … `except ValueError as ve:`, and the rules are the same. Book: Section 5.5.4.

Three Details That Make It Work

`s.next()` **discards the bad token.** A failed `nextInt()` does not consume `b`; without `next()`, the loop would fail on the same `b` forever.

Do not build a new `Scanner` in the `catch`. The old one may already have buffered the input; with a redirected file the new one often finds nothing (Exercise 5.12).

`while (s.hasNext())` ends gracefully if the input stops before `99`.

```
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]
```

The Python version, `for line in sys.stdin`, stops cleanly at the end of input, and the bad text is already consumed when `int()` fails; its output is shown above. Book: Section 5.5.4.

Progressively Bigger Nets

Listing 5.18 divides 100 by numbers taken from `{"12", "x7", "0"}` at positions 0, 1, 2 and 5.

Listing 5.18 The loop of `CatchOrder.java`: three `catch` clauses, from the most specific to the most general.

```
        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);
            }
        }
```

Position 0: $100/12 = 8$. *Position 1:* `"x7"`, *the first net. Position 2:* `100 / 0`, *the widest net. Position 5: no such position, the second net.* `catch (Exception e)` *first would make the others unreachable, and Java refuses to compile it. Multi-catch:* `catch (A | B e)`*; Python:* `except (ValueError, ZeroDivisionError) as e:`*. Book: Section 5.5.5.*

`finally`: the Block That Always Runs



Figure 5.6 The paths through a `try`–`catch`–`finally` statement. Every path passes through a `finally` block.

***Figure 5.6**: after a normal end, after a `catch`, when an exception escapes, and even after `return` — every path passes through `finally`. That is where files are closed and machines are switched off. Book: Section 5.5.6.*

Worked Example 1: `finally` and `return`

Listing 5.19 has a `try` and a `finally` but no `catch`.

Listing 5.19 The method `divide` of `FinallyDemo.java`: `finally` runs after a normal `return` and when an exception escapes.

```
    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");
        }
```

```
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.
```

Case 1: 5 is computed, `finally` prints, then `main` receives 5. Case 2: `1 / 0` throws, `finally` prints on the way out, then `main`'s `catch` handles it; `result =` is never printed. Book: Section 5.5.6.

Worked Example 1, Continued: Trace It

Exercise 5.5: `f(x)` prints `A`, computes `10 / x`, prints `B`, returns `y`; `catch` prints `C` and returns −1; `finally` prints `D`.

`f(0)`: `A`; `10 / 0` throws, so `B` is skipped; `catch` prints `C` and prepares −1; `finally` prints `D`; returns −1.

`f(1)`: `A`; `y = 10`; `B`; `return` prepares 10; `finally` prints `D`; returns 10.

Python adds `else`, which runs only when the `try` raised nothing; the order is `try`, `except`, `else`, `finally`.

The `finally` block runs after the value to be returned has been computed and before it is actually returned. Book: Section 5.5.6 and Solution 5.5.

Throwing an Exception Yourself

Listing 5.20 forbids a grayscale value above 255 with `throw new IllegalArgumentException(…)`; Python writes `raise ValueError(…)`.

`main` calls it with 270 inside a `try`; the `catch` prints "270 is an invalid Grayscale value", resets the value to 255, and processing continues.

Listing 5.20 The method `isExceeded` of `IllegalArgumentExceptionExample.java`: throwing `IllegalArgumentException` for a value above 255.

```
    public static boolean isExceeded(int x) {
        boolean result = false;
        if (x > 255) {
            throw new IllegalArgumentException("Value exceeds 255");
        } else {
            result = true;
        }
        return result;
    }
```

The method's name lies: `isExceeded` returns `true` when the value is fine. Call it `checkGrayscale`, return nothing, and throw on bad input. Throw only for errors, never as a clever way out of a loop. Book: Section 5.5.7.

Unchecked Exceptions: Our Own Mistakes

Listing 5.21 provokes the three most famous ones; all extend `RuntimeException`.

Listing 5.21 The body of `UncheckedDemo.java`: three unchecked exceptions caused by programming mistakes.

```
        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);
        }
```

*Index 6 of a length-6 array; `length()` on a `null` slot; `(String)` on an `Integer`. These are the program's fault: the right response is to **fix the code**, not to catch them. Python's equivalents: `IndexError`, `TypeError`, `AttributeError`. Book: Section 5.6.1.*

Checked Exceptions: the Compiler Insists

A missing file, a full disk, a lost network drive: nobody's programming error, and no care can prevent them. A correct program can only **plan** for them.

`new FileReader("filename.txt")` alone does not compile:

```
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
```

Checked exceptions extend `Exception` but not `RuntimeException`: the blue boxes of Java's family tree. Book: Section 5.6.2.

Catch It, or Declare It

Listing 5.22: `firstLine` **declares** `throws IOException`, because it cannot know what a missing file should mean; `main` **catches**, because it knows it wants a message.

Listing 5.22 From `OpenFile.java`: `firstLine` declares `IOException` with `throws`; `main` catches it.

```
    static String firstLine(String fileName) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(fileName));
        String line = br.readLine();
        br.close();
        return line;
    }
            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.
```

Specific net first. `throws` on `main` would compile, and crash the program. Python has no checked exceptions: it documents them in a `Raises:` section of the docstring instead. Book: Sections 5.6.2–5.6.4.

Exceptions of Your Own

A custom exception says **what kind** of problem (its class), **what exactly** (its message), and **with which data** (its fields and its cause). **Listing 5.23** is one.

Listing 5.23 The nested class `InvalidWeightException` in `WeightCheck.java`: a checked exception that carries the offending text.

```
    // 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;
        }
    }
```

Checked (extends `Exception`) because bad data from outside is an expected outcome; unchecked (`RuntimeException`) would signal a programming mistake. `super(message, cause)` reuses what `Exception` already stores. Book: Section 5.7.

Chaining: Wrap, Don't Lose

In **Listing 5.24** the `NumberFormatException` becomes the **cause** of our exception: one laundry-level message, nothing lost.

Listing 5.24 The method `parseWeight` of `WeightCheck.java`: translating a low-level exception, and rejecting a weight outside 0–50 kg.

```
    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;
    }
```

For −1 and 80 there is no underlying exception, so the cause is `null`. Python writes `raise InvalidWeightError(…) from ve`, and the original is `e.__cause__`. Book: Section 5.7.

Closing What You Open, the Old Way

Every open file holds an operating-system handle, and written data is only guaranteed on disk after `close()`. **Listing 5.25** closes in `finally`.

Listing 5.25 The `finally` block of `CopyFileClassic.java`: closing both streams, the pre-Java-7 way.

```
        } 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 */ }
        }
```

Both streams are declared before the `try` and set to `null`; `finally` checks for `null` and wraps each `close()` in its own `try`. Nearly half the program is about closing. Book: Section 5.8.2.

`try`-with-resources

Anything `AutoCloseable` opened in the parentheses after `try` is closed when the block ends, however it ends, in reverse order. **Listing 5.26** is the modern copy.

Listing 5.26 The method `copy` of `CopyFile.java`: `try`-with-resources closes both streams automatically.

```
    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;
    }
```

```
Copied 102 bytes from notes.txt to notes_copy2.txt
Could not copy nothing.txt: nothing.txt (No such file or directory)
```

`copy` declares, `main` catches. Python's `with open(a, "rb") as source, open(b, "wb") as target:` gives the same guarantee. In new code, open every file this way. Book: Section 5.8.3.

Writing and Appending Text

`FileWriter` replaces a file; `FileWriter(name, charset, true)` **appends**. Python: `open` with mode `"w"` or `"a"`.

Build text once: `text = text + more` in a loop is the quadratic trap of Equation (5.2); use `StringBuilder`, or collect pieces and `"".join(parts)`.

Say the encoding: `StandardCharsets.UTF_8` and `encoding="utf-8"`, or "José" written on one machine may be garbled on another.

Read to the end: `while ((line = in.readLine()) != null)` in Java; `for line in f` in Python.

Streams carry bytes, readers and writers carry characters, and a `BufferedReader` hands out whole lines. The receipt program writes three items and a total, appends "Thank you for choosing Profitina Laundry!", and reads the file back with line numbers. Book: Sections 5.8.1 and 5.8.4.

CSV: the Two Traps of `split`

Listing 5.27 prints every field between `<` and `>` so that spaces and empty fields become visible.

Listing 5.27 From `CsvSplit.java`: the first trap of splitting a CSV line in Java.

```
        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));
```

```
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><>
```

Trap 1 (Java only): `split(",")` silently drops trailing empty fields; `split(",", -1)` keeps them. Trap 2 (everyone): a quoted field such as `"Santoso, Budi"` contains a comma; Python's `csv.reader` handles it, Java needs a library. Book: Section 5.8.5.

Profitina Laundry: the Pipeline



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.

Figure 5.7*: a bad line is local, so we record it and continue; a missing file or a wrong header is global, so the loader throws and the application decides. Book: Section 5.9.*

Worked Example 2: a Damaged File

`orders.csv`, damaged in seven ways, with 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
```

Rules: exactly five fields; non-empty id and name; a known service; a weight in $(0,50]$ *kg;* `yes` *or* `no`*; no repeated id. Book: Section 5.9.*

The Price, in Whole Rupiah

`WASH` Rp 7,000, `IRON` Rp 5,000, `WASH_IRON` Rp 10,000 per kilogram; express costs 50% more (Equation (5.4)).

$$p = \begin{cases} \text{round}(w \cdot r) & \text{normal order,} \\ \text{round}(w \cdot r) + \lfloor \text{round}(w \cdot r)/2 \rfloor & \text{express order.} \end{cases} \tag{5.4}$$

Sari: round$(3.5 \times 7000) = 24{,}500$. Budi, express wash-and-iron: $50{,}000 + 25{,}000 = 75{,}000$.

Money is computed in integers: $0.1 + 0.2$ is not exactly 0.3 in binary floating point.

`Order` keeps the services and rates in two parallel arrays (Java) or a dictionary (Python), and `rateFor` returns -1 for an unknown code. Book: Sections 5.9 and 5.9.1.

The `Order` Class Knows Its Price

Listing 5.28 implements Equation (5.4) with a lookup in two parallel arrays.

Listing 5.28 From `Order.java`: the rates, `rateFor`, and `price`.

```
    static final String[] SERVICES = {"WASH", "IRON", "WASH_IRON"};
    static final long[] RATES = {7000, 5000, 10000};
    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
    }
```

`Math.round` returns a `long`; `base / 2` is integer division. Arrays suit the services because their number is fixed when the program is written; Python uses a dictionary. Book: Section 5.9.1.

`parseLine`: One Rule at a Time

Listing 5.29: count the fields **before** using any; the first failing rule throws `OrderFormatException` with the line number.

Listing 5.29 Part of `parseLine` in `OrderCsv.java`: the field count first, and the weight wrapped as a cause.

```
    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);
        }
        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");
        }
```

Between the two parts, the fields are trimmed and the id, name and service are checked the same way. `!(kg > 0 && kg <= 50)` also rejects `NaN`, for which every comparison is false. Book: Section 5.9.3.

`load`: the Inner `try` Is the Design

Listing 5.30: after the header check, each line has its own `try`, whose `catch` records the message and **keeps going**.

Listing 5.30 The loop of `load` in `OrderCsv.java`: one bad line becomes one message.

```
        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
            }
        }
```

The file is opened with `try`-with-resources, and a wrong header throws before this loop. The position of the inner `try` is the design decision of the pipeline expressed in code. A duplicate id is thrown inside the same `try`, so every bad line is reported the same way. Book: Section 5.9.3.

The Same Loop in Python

Listing 5.31 reads through `csv.reader`, whose `line_num` gives the number of the line just read.

Listing 5.31 The body of `load` in `order_csv.py`: the same loading and validation in Python.

```
    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
```

Files are opened with `newline=""`, as the `csv` documentation requires, and `encoding="utf-8"`. Book: Section 5.9.3.

Worked Example 2: the Result

`LaundryApp.java` loads, reports, adds Maya's order, prices, saves, reloads, and then tries a missing file and a foreign one:

```
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).
```

Revenue $24{,}500 + 75{,}000 + 10{,}000 + 126{,}000 + 22{,}500 = 258{,}000$*. The Python application prints the same lines and writes a byte-identical* `orders_clean.csv`*. Book: Section 5.9.4.*

Profitina in Practice

Lists of unknown size: a site scan does not know how many pages, structured-data blocks or signals it will find; every one is a growable list.

CSV at the edges: the weekly score-trend export and the personal data export (UU PDP 27/2022) use UTF-8, a header row and proper quoting.

Failures reported, not hidden: when the local model (Qwen2.5 on Ollama) is unavailable, the application *says so*; a release reports success only after its health check passes.

Validation before trust: every community link is screened; a post that fails to send is restored for another try.

The software advises, the user decides — and it can only advise well if it tells the truth about what went wrong. Book: Section 5.10.

Common Mistakes

`i <= a.length`, a hard-coded length, or mixing `length`, `length()` and `size()`.

An alias taken for a copy; `==` on arrays in Java; `remove(3)` on a list of `Integer`; expecting −1 from Python's `index`.

The leftover line break after `nextInt()`; retrying `nextInt()` without discarding the bad token.

`catch (Exception e)` **first or everywhere**; an empty `catch`; catching an unchecked exception instead of fixing the bug; `throws` on `main` to silence the compiler.

A file never closed; the default encoding; `split(",")` losing trailing fields; a method whose name lies; exceptions used for ordinary control flow.

Nineteen mistakes are listed in the book, each with its fix. Book: Section 5.11.

Summary: Five Things to Remember

1. An array is a fixed row of slots indexed 0 to $n-1$, reached directly by Equation (5.1); the variable holds a reference, so assignment makes an alias.

2. A list grows by itself; geometric growth makes appends cheap on average (Equation (5.3)). Java lists of numbers need `Integer`.

3. An exception is thrown, propagates back through the calls, and is caught by the first matching clause; order the nets from specific to general; `finally` always runs.

4. Unchecked exceptions are bugs to fix; checked exceptions must be caught or declared. Python has none, so document them.

5. Close every file with `try`-with-resources or `with`, say UTF-8, and validate every CSV field.

Profitina Laundry now loads a damaged file, explains every bad line, keeps going, and saves a file that reloads cleanly. Book: Section 5.12.

Next Time

Chapter 6 — Documentation, Testing, and Interfaces. Javadoc and docstrings, JUnit and `unittest`, overloading, and the interface as a contract.

Profitina Laundry's prices become a family of interchangeable price rules, each tested.

Before then: work Exercises 5.2, 5.5, 5.8 and 5.10, in Java and in Python.

The `Raises:` docstring sections met today are a first taste of Chapter 6's documentation. Book: Sections 5.13 and 5.14.

Lecture 6 — Documentation, Testing, and Interfaces

Course: Object-Oriented Programming (EF234302)

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*, Chapter 6

Lecturer: Irfan Subakti — yifana@gmail.com

Today our classes learn to make promises in writing — for people, for test programs, and for the compiler. Book: Chapter 6, Sections 6.1–6.19.

What This Meeting Covers

1. Documentation — several constructors, Javadoc and docstrings, `javadoc`, `pydoc` and `doctest`.

2. Testing — a test with no library, arrange–act–assert, JUnit 6, `unittest` and pytest, boundaries, test first.

3. Equality — identity versus equality, `equals` with `hashCode`, `toString`, `__repr__` and `__str__`.

4. Overloading — how Java chooses a version, and what Python does instead.

5. Interfaces — `Measurable`, swapping a representation, `Comparable`, listeners, ABCs and protocols.

6. Profitina Laundry — five price rules behind one interface, every rule tested.

The three big ideas answer three awkward questions, in order; the smaller tools in between are what they rely on. Book: Section 6.1.

Where We Left Off: Three Awkward Questions

Chapter 5 gave us arrays, lists, input, exceptions and files: enough to run a small shop for a morning, but not yet trustworthy.

How does a colleague know how to use a class? → documentation.

How do we know it works? → testing, every time, in a second.

How do we change it without breaking its users? → interfaces: the rest of the program depends on a promise, never on the details.

> **KEY INSIGHT.** Documentation, tests and interfaces are three ways of writing down a *promise*: in words for people, in code that checks it, and in the type system.

Good software keeps all three in step. Book: Section 6.1.

Two Ways to Open an Account

Opened with money: overdraft 50. Opened empty: overdraft 10. Java gives the class **two constructors**, told apart by their parameter lists (Listing 6.1).

Listing 6.1 The two constructors of `BankAccount.java`.

```
    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;
    }
```

```
BankAccount[balance=0.0, overdraft=10]
BankAccount[balance=25.0, overdraft=50]
```

`new BankAccount()` picks the constructor with no parameters; `new BankAccount(25)` picks the one with a `double`. A class with no constructor gets an empty one from the compiler. Book: Sections 6.2 and 6.2.1.

Python: One `__init__`, a Default Argument

Python allows only one `__init__` per class; a second would replace the first. Listing 6.2 uses the default `None`, "nothing here".

Listing 6.2 The `__init__` of `bank_account.py`: one method with a default argument does the work of Java's two constructors.

```
    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
```

Why `None` and not `0`? `BankAccount(0)` is a deposit that happens to be zero and earns the larger overdraft. One constructor can also call another in Java with `this(...)` (Exercise 6.3). Book: Section 6.2.1.

Comments for Someone Who Never Reads Your Code

> **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 item 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.

You have used `String` and `ArrayList` for five chapters without seeing their source: the API pages told you everything a caller needs.

An ordinary comment explains a line to someone reading the code; a documentation comment explains the contract to someone using it. Book: Section 6.3.

A Javadoc Comment, Tag by Tag

The first sentence is the *summary*; then one `@param` per parameter, `@return`, and `@throws` (Listing 6.3).

Listing 6.3 The method `interestOn` of `BankAccount.java`, with its Javadoc comment.

```java
    /**
     * 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;
    }
```

The body computes compound interest, $P(1+r)^n - P$: $123\,600$ on $1\,000\,000$ at 6% for 2 years. "6% is written 0.06" belongs in the comment; "uses `Math.pow`" does not. Python: `Args:`, `Returns:`, `Raises:`. Book: Sections 6.3.1 and 6.3.2.

The Tags You Will Use

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

Table 6.1: every Javadoc tag, and the place where the same information goes in a Python docstring. Book: Section 6.3.1.

A Trap: `/` Instead of `*/`

End the comment of `interestOn` with a lonely `/` and the comment runs on to the **next** `*/` — swallowing the whole method. `javac` says nothing; `javap` shows the method is gone:

```
Compiled from "Interest.java"
public class Interest {
  public Interest();
  public double twice(double);
}
```

A program that calls it is then refused with `cannot find symbol`.

An editor with syntax coloring shows the whole method in comment color — one more reason to use one. Book: Section 6.3.1.

Examples That Check Themselves

The class docstring of `bank_account.py` contains lines that look like a Python session:

Listing 6.4 From the class docstring of `bank_account.py`: two doctest examples.

```
    >>> BankAccount().overdraft
    10
    >>> BankAccount(25).overdraft
    50
```

`python3 -m doctest -v bank_account.py` runs every example and compares; Listing 6.4 can never go out of date without somebody noticing.

```
3 tests in 7 items.
3 passed and 0 failed.
Test passed.
```

A doctest is our first glimpse of testing: code that runs other code and compares the result with an expectation. Book: Section 6.3.3.

From Comments to Pages

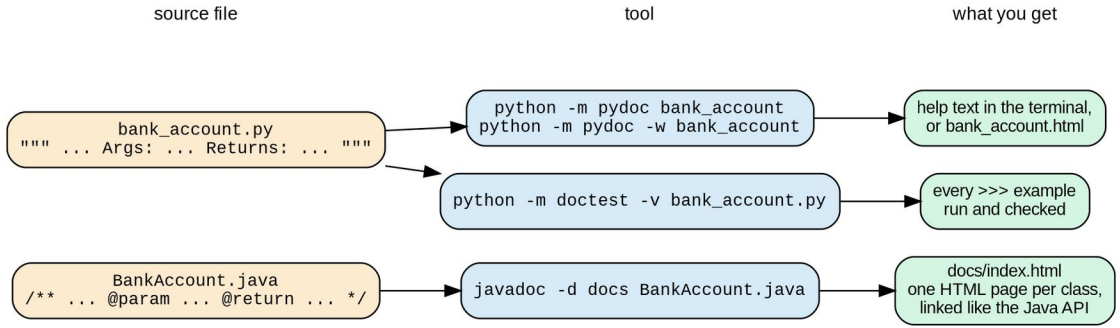


Figure 6.1 From documentation comments to documentation: `javadoc` builds linked HTML pages, `pydoc` prints help text or writes an HTML page, and `doctest` runs the examples written in docstrings.

Figure 6.1: `javadoc -d docs BankAccount.java` builds pages like the Java API and reports a misspelled or missing `@param`; `python3 -m pydoc bank_account` prints help. Book: Sections 6.4.1 and 6.4.2.

Reading the API: `Random` and the Seed

The API page of `java.util.Random` says `nextInt(10)` returns 0 (inclusive) to 10 (exclusive), so a machine from 1 to 4 is `nextInt(4) + 1` (Listing 6.5).

Listing 6.5 From `RandomDemo.java`: the main methods of `java.util.Random`, with a fixed seed.

```
        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.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);
        }
```

```
Machines for 8 loads: 1 3 3 4 1 1 4 4
```

The seed `2026` makes every run print the same numbers. A real laundry wants no seed; a test wants one, because a test whose input changes on every run proves nothing. Python: `random.Random(2026)`, `randint(a, b)` includes b. Book: Section 6.4.3.

What a Unit Test Is

> **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.

Looking is unreliable — a person misses `27000` for `28000`; code never does.

Looking is not repeated — a suite runs again after every change.

Tests change how you design — hard-to-test code is almost always badly divided.

A program is correct when it does what its documentation promises for every input it accepts; we cannot try them all, so we choose cleverly. Book: Section 6.5.

The Class Under Test

A `Dog` has a name and an age; a birthday adds one year; a negative age is refused (Listing 6.6).

Listing 6.6 The constructor and `incrementAge` of `Dog.java`, the class we are going to test.

```
    public Dog(String name, int age) {
        if (age < 0) {
            throw new IllegalArgumentException("age must not be negative: " + age);
        }
        this.name = name;
        this.age = age;
    }
    public void incrementAge() {
        age++;
    }
```

`Dog` has no `main`: it is a library class, and the first code to use it will be its tests. The Python `dog.py` has the same shape, with read-only properties. Book: Section 6.5.1.

Worked Example 1: a Test with No Library

All a test needs is "I expected this, I got that — do they agree?" Listing 6.7 writes that as `check`.

Listing 6.7 The method `check` of `DogCheck.java`: testing without any test library.

```
    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);
        }
    }
```

The expected value comes first, and a failure says what was expected and what actually happened; most of the value of a test lies in that message. Book: Section 6.5.2.

Worked Example 1: the Checks

Listing 6.8 builds Rover aged 5, acts, and checks against numbers worked out **before** running anything.

Listing 6.8 From the `main` of `DogCheck.java`: three checks and one edge case.

```
        // 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");
        }
```

*The last check is an **edge case**: it passes only when `new Dog("Rex", -1)` throws. `main` ends with `System.exit(1)` if anything failed, because build tools read the exit status. Book: Section 6.5.2.*

Worked Example 1: Break It on Purpose

Unbroken, both languages print four `PASS` lines and `4 passed, 0 failed`. Now change `age++` to `age += 2` and run again:

```
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
```

The exit status is 1, and the message names the broken promise.

> **THINK.** Never use `assert` to check *user data*: `python -O` removes every `assert`, and Java's `assert` is off unless you run `java -ea`. `assert` is for tests.

Undo the change before you continue. Book: Section 6.5.2.

Arrange, Act, Assert

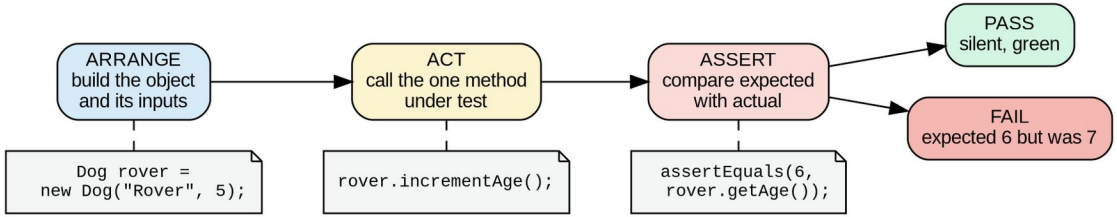


Figure 6.2 Arrange, act, assert: build the object, call the one method under test, compare the expected value with the actual one.

Figure 6.2: arrange minimally, act once (one test, one behavior), assert against a literal — 6, never a copy of the method's own formula, which agrees with the same bug. Book: Section 6.6.

Comparing Numbers with a Fraction

0.1 is stored as a binary fraction slightly different from one tenth, and the errors add up (Listing 6.9).

Listing 6.9 From `DoubleEquality.java`: why `==` is the wrong test for two doubles.

```
    static boolean close(double expected, double actual, double epsilon) {
        return Math.abs(expected - actual) <= epsilon;
    }
        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));
```

```
0.1 + 0.2          = 0.30000000000000004
a == 0.3           : false
close(0.3, a, 1e-9): true
```

Accept any answer close enough: Equation (6.1), with $|0.3 - 0.30000000000000004| = 4 \times 10^{-17} \le 10^{-9}$.

$$|e - a| \le \varepsilon \tag{6.1}$$

JUnit `assertEquals(e, a, delta)`, `assertAlmostEqual(e, a, delta=…)`, pytest `a == pytest.approx(e)`. Better still for money: whole rupiah in a `long` or `int`. Book: Section 6.6.1.

JUnit 6: the Same Tests, Framed

Listing 6.10 marks tests with `@Test`; `@BeforeEach` builds a fresh Rover before every test.

Listing 6.10 From `DogTest.java`: the `Dog` tests written for JUnit 6.

```
    @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");
    }
```

No `main`: the runner has it. Expected value first. The third test, `assertThrows(…, () -> new Dog("Rex", -1))`, hands JUnit a lambda to run inside its own `try`. Book: Sections 6.7.1 and 6.7.2.

`unittest`: the Java Shape in Python

In Listing 6.11 a method whose name starts with `test` is a test; `setUp` plays `@BeforeEach`; `with self.assertRaises(...)` replaces the lambda.

Listing 6.11 From `test_dog.py`: the `Dog` tests written for `unittest`.

```
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)
```

Run with `python3 -m unittest -v test_dog`; with no file name it discovers every `test*.py`. A failure prints `.F.` first: a dot per pass, `F` per failure. Book: Section 6.7.3.

JUnit 6, `unittest`, pytest

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`

Table 6.2: pytest drops the ceremony — plain functions, plain `assert` — and prints the value of every piece of a failing expression. Book: Section 6.7.4.

Equivalence Classes and Boundaries

Wash and fold: Rp 7 000 per **started** kilogram, never less than 3 kg. Table 6.3 picks one representative per class and the values at each border.

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)

Boundaries are where programmers write `<` for `<=`. For an array, test the first, the last and the empty; for every `if`, both branches; for every promised exception, a test that provokes it. Book: Section 6.8.1.

Test First: Red, Green, Refactor

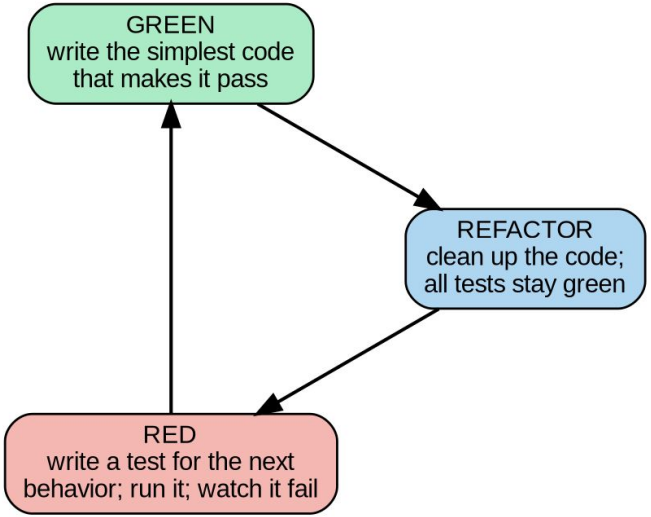


Figure 6.3 The test-first cycle: a failing test (red), the simplest code that passes (green), then tidying with the tests as a safety net (refactor).

Figure 6.3: write one failing test and watch it fail; write the simplest code that passes; then tidy with the tests as a safety net. Book: Section 6.8.2.

When Are Two Objects Equal?

Monday: “Sari”, `0812-555-0101`. Thursday: ” Sari “, `0812 555 0101`. The same customer? If the class says nothing, the answer is **no**, and her loyalty points split in two.

> **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)`).

The `equals` every class inherits from `Object` is simply `==`: an object is equal only to itself. Python’s default `__eq__` behaves the same way. Book: Sections 6.9 and 6.9.1.

Identical or Equal?

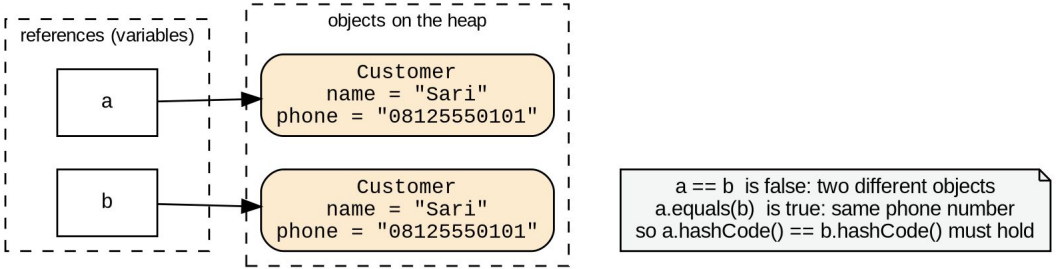


Figure 6.4 Two references to two different objects with the same contents: not identical, but equal if the class says so — and then their hash codes must be equal too.

Figure 6.4: two variables, two separate objects, identical contents — not identical, but equal if the class says so, and then their hash codes must be equal too. Book: Section 6.9.1.

`equals` Without `hashCode`

`HalfCustomer` overrides `equals` and forgets `hashCode`. Listing 6.12 shows what a list and a hash set make of it:

Listing 6.12 From `IdentityDemo.java`: a class with `equals` but no `hashCode`.

```
        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());
```

```
  c.equals(d) : true
  list contains d: true
  set contains d : false
  set size       : 2
```

*A hash set first asks for a **hash code** to choose a bucket, and only compares with `equals` inside it; the inherited `hashCode` sends `c` and `d` to different buckets. `javac -Xlint:all` warns about exactly this. Book: Section 6.9.1.*

The Rule That Makes Hashing Work

For any two objects a, b and hash function h, Equation (6.2) must hold:

$$a \text{ equals } b \Rightarrow h(a) = h(b) \tag{6.2}$$

Not backwards: two different objects may share a code (a *collision*), which costs only a little speed.

> **RULE.** Whenever you override `equals`, override `hashCode` too, using the same fields.

Python is stricter, which is kinder: a class with `__eq__` and no `__hash__` becomes unhashable, and putting it in a set raises `TypeError` on the spot. Book: Section 6.9.1.

`Customer`, Done Right

A customer is identified by the **digits** of the phone number; Listing 6.13 normalizes it in the constructor and compares nothing else.

Listing 6.13 `equals` and `hashCode` of `Customer.java`: equality by phone number, and a hash of the same field.

```
    @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);
    }
    @Override
    public int hashCode() {
        return phone.hashCode();
    }
```

The four steps: same object? right class (and not `null`)? cast; compare the fields with `equals`. Never write `equals(Customer other)`: that overloads instead of overriding, and `@Override` catches it. Book: Section 6.9.2.

The `equals` Contract

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 (6.2): equal objects have equal hash codes.

```
a == b      : false
a.equals(b) : true
same hash   : true
distinct customers: 2
visits by Sari: 3
```

The output of `Customer`'s `main`: equal but not identical, the same hash, two distinct customers in a set, and Sari's three visits counted together. Python returns `NotImplemented` for a foreign type. Book: Section 6.9.2.

How an Object Describes Itself

Java calls `toString()` for `println(x)` and `"…" + x`; the default, `PlainCustomer@1b6d3586`, tells a human nothing. Convention: `Customer[name=Sari, phone=08125550101]`.

Python splits the job: `__repr__`, unambiguous and for developers, `Customer(name='Sari', phone='08125550101')`; `__str__`, friendly and for users, `Sari (08125550101)`.

A Java **record** or a Python `@dataclass(frozen=True)` writes all of these for you — but compares *every* field.

> **TRAP.** Never put secrets into `toString` or `__repr__`: they end up in logs and error reports (UU PDP 27/2022).

A list prints its items with `__repr__`. Java has only `toString`, so a customer-facing form gets its own method, such as `receiptLine()`. Book: Sections 6.9.2 and 6.10.

One Name, Several Parameter Lists

> **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.

The two `BankAccount` constructors are overloads; `println` has ten. The **return type is not part of the signature**: `javac` refuses two methods that differ only in it.

Two `average` methods, for accounts and for countries, share a name — and a nearly identical body. Overloading lets us name them once, not write them once. Book: Section 6.11.

Worked Example 2: Which Version Runs?

Listing 6.14: three versions of `describe`, six calls. Predict each line before you read it.

Listing 6.14 The three overloads of `OverloadDemo.java`: which version does each call choose?

```
    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";
    }
```

```
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
```

Exact match first; then ***widening****:* `byte` → `int`, `long` → `double` *(so* `3L` *becomes* `3.0 kg`*),* `char` → `int` *(so* `'A'` *becomes 65 pieces); then boxing; then varargs. Book: Section 6.11.1.*

Python Has No Overloading

A second `def average` simply **replaces** the first, like a second `x = …`; the accounts then fail at run time.

Python's tools, in Listing 6.15: default arguments, `*args`, and `functools.singledispatch`, which chooses by the type of the first argument at run time.

Listing 6.15 From `overload_demo.py`: overloading by type with `functools.singledispatch`.

```
@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"
```

`describe(True)` picks the `int` version, because `bool` is a subclass of `int` — the same trap as Java's `char`. Overload only when the versions do the same thing. Book: Section 6.11.2.

Interfaces

What does `average` really need from each object? Not that it is an account or a country: only **a number to add up**.

> **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*.

Listing 6.16 `Measurable.java`, without its comment: an interface with a single method and no code.

```
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
}
```

Listing 6.16: interface methods are automatically `public` and `abstract`. An interface lives in its own file, like a public class. Book: Section 6.12.

One `average` for Every `Measurable`

In Listing 6.17 the parameter's type is the interface, so the loop may call only `getMeasure()` — all it needs.

Listing 6.17 From `MeasurableDemo.java`: `Account`'s promise kept, and one `average` for all.

```
    @Override
    public double getMeasure() {
        return balance;
    }
    public static double average(Measurable[] objects) {
        if (objects.length == 0) {
            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;
    }
```

*When `obj` is an `Account`, `Account`'s version runs; when a `Country`, `Country`'s. That choice at run time is **dynamic dispatch**, the heart of polymorphism (Chapter 7). Book: Section 6.12.*

The Mean, and a Broken Promise

The loop computes the ordinary mean Equation (6.3); for three accounts, $(100+150+125)/3=125$.

$$\acute{x}=\frac{1}{n}\sum_{i=1}^{n} x_i \tag{6.3}$$

Delete `getMeasure` from `Country`, and the compiler catches it before the program ever runs:

```
MeasurableDemo.java:20: error: Country is not abstract and does not override abstract method
getMeasure() in Measurable
class Country implements Measurable {
^
1 error
```

`new Measurable()` is refused too: an interface is a contract, not a thing. A variable may still have the interface type: `Measurable y = new Account(…)`. Book: Section 6.12.

Python: a Protocol, Checked by Shape

In Listing 6.18, `Account` and `Country` never mention `Measurable`, yet `isinstance(accounts[0], Measurable)` is `True`: they have the method.

Listing 6.18 From `measurable_demo.py`: the `Measurable` protocol and one `average`.

```
@runtime_checkable
class Measurable(Protocol):
    """Anything that can be reduced to one number for averaging."""

    def get_measure(self) -> float:
        ...
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)
```

The `...` means "no body here". Python ignores the annotation `Sequence[Measurable]` when running; tools such as mypy read it. Book: Section 6.12.

Swapping a Representation

A tiny table of integer keys can live in an unsorted list (`LinearDB`) or a binary search tree (`TreeDB`); the client chooses at start-up.

Without an interface: two variables, one always `null`, and a flag tested at every operation. A third representation means a third branch in every one of those places.

With an interface: one variable of type `DBops`, chosen once; every call is an ordinary call, and dynamic dispatch runs the right code.

*`DBops` is a **supertype** of both classes; `db = new TreeDB()` converts up automatically, because a `TreeDB` offers everything `DBops` promises. Book: Sections 6.13.1–6.13.3.*

The Client Depends Only on the Interface

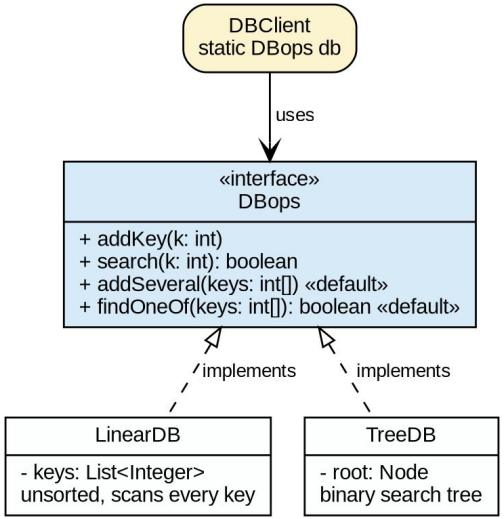


Figure 6.5 The client depends only on the interface `DBops`; `LinearDB` and `TreeDB` implement it and can be exchanged without changing the client.

Figure 6.5 in UML: «interface» above a name marks an interface; a dashed line with a hollow triangle means "implements"; a plain arrow means "uses". Book: Section 6.13.2.

Default Methods: Shared Code in an Interface

Listing 6.19: `addSeveral` and `findOneOf` are written **once**, from the two abstract methods, and inherited by both classes.

Listing 6.19 From the interface `DBops` in `dbiface/DBClient.java`: two abstract methods and a default method.

```
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);
        }
    }
```

`findOneOf` is a second default method, a loop over `search`. An interface has no fields: it describes behavior, not state. Python writes the same interface as an abstract base class, `DBops(ABC)`, and `DBops()` raises `TypeError`. Book: Section 6.13.4.

`Comparable`: Only the Sign Is Promised

`Arrays.sort` asks "should this come before that?" through `compareTo`, whose meaning is Equation (6.4):

$$\operatorname{sign}(\texttt{a.compareTo(b)}) = \begin{cases} -1 & \text{if } a \text{ comes before } b \\ 0 & \text{if they are equal in order} \\ +1 & \text{if } a \text{ comes after } b \end{cases} \tag{6.4}$$

`"Ace".compareTo("Zonk")` is -25, not -1. A `switch` with cases only for $-1, 0, 1$ is therefore wrong.

`List`, `Iterable` and `AutoCloseable` are interfaces you have already relied on: `List<Order> orders = new ArrayList<>()`, the for-each loop, try-with-resources. Book: Sections 6.14 and 6.14.1.

Worked Example 3: a Sort That Lies

Listing 6.20: `SwitchName` leaves `result` at `Integer.MIN_VALUE` for ±24, so each Bogdanov "comes before" Zonk *and* after it.

Listing 6.20 The `switch` in `SwitchName.compareTo` of `NameSortDemo.java`: trusting `String.compareTo` to return only −1, 0 or 1.

```
        switch (lastName.compareTo(o.lastName)) {
            case -1:
                result = -1;
                break;
            case 1:
                result = 1;
                break;
            case 0:
                result = firstName.compareTo(o.firstName);
                break;
        }
```

```
[Ace, Portgas D.]
[Bogdanov, Aleksandra]
[Zonk, Godot]
[Bogdanov, Alexei]
```

The correct `Name` looks only at whether the comparison is zero: if the last names differ, their comparison is the answer, whatever number it is. With three names the bug hides; with four it shows. Book: Section 6.14.1.

Casting Up and Down

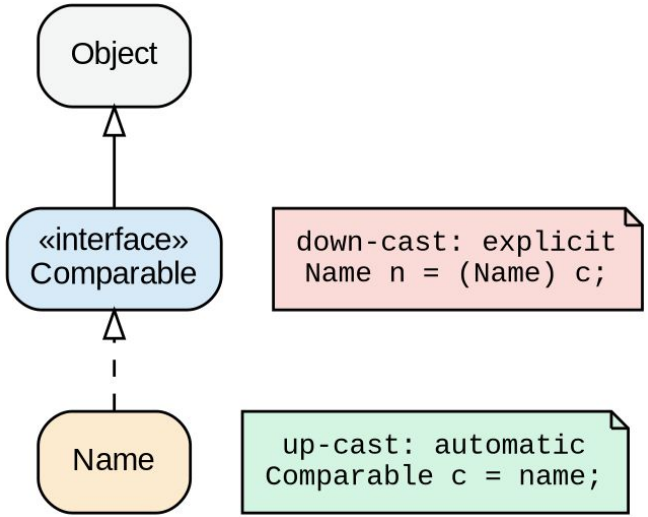


Figure 6.6 Moving up the type hierarchy (from `Name` to `Comparable` or `Object`) is automatic; moving down needs an explicit cast, which is checked when the program runs.

Figure 6.6: up-casting is automatic; down-casting, `(SwitchName) other`, is checked at run time and may throw `ClassCastException`. The generic `Comparable<Name>` needs no cast. Python sorts with `__lt__` or a `key=` function. Book: Section 6.14.1.

Listeners: a Button That Knows Only an Interface

A button keeps a list of listeners and, when clicked, calls each one (Listing 6.21); `BouncingBall`, `Pendulum` and `RotatingSpiral` each just write `implements StopListener`.

Listing 6.21 From `StopButtonDemo.java`: the listener interface and the button that knows only it.

```
/** Anything that wants to hear when the Stop button is pressed. */
interface StopListener {
    void stopRequested(String 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);
    }
```

This is the idea behind `ActionListener` and every Swing button (Chapter 10). In Python a plain function can be a listener, which is what Tkinter's `command=` does. Book: Section 6.14.2.

Four Ways to State a Contract

Table 6.4 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

Table 6.4: an ABC fails loudly when an incomplete object is created; a Protocol is checked by a type checker such as mypy before running. Book: Section 6.15.

Pricing at Profitina Laundry

Wash and fold, wash and iron: per started kilogram, 3 kg minimum. Dry cleaning: per piece. Shoes: one fixed price. Express: 50% more, whatever the service. Members: 10% off, whatever the service.

> **KEY INSIGHT.** *Program to an interface, not an implementation.* Code that needs a price asks for "something that can price an order" — then new ways of pricing are new classes, and the code that uses prices never changes.

Next month a rainy-season promotion, next year a hotel price: with one `if` per rule, every new rule means editing the receipt code. Book: Section 6.16.

The Contract: `PriceRule`

Listing 6.22: the documentation is part of the contract — whole rupiah, never negative, refusal by `IllegalArgumentException`.

Listing 6.22 The interface `PriceRule.java`, without its static formatting helper.

```java
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();
```

`PriceRule.rupiah(28000)` formats `Rp 28.000`. In Python `PriceRule` is an abstract base class, so an incomplete rule fails the moment it is created. Book: Section 6.16.2.

The Per-Kilogram Rule

Rate r, minimum m, weight w: Equation (6.5), where $\lceil 3.2 \rceil = 4$; Listing 6.23 is the formula line for line.

$$P_{\text{kilo}} = r \cdot \max(m, \lceil w \rceil) \tag{6.5}$$

Listing 6.23 The method `priceOf` of `PerKiloRule.java`: price by started kilogram, with a minimum.

```
    @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;
    }
```

$w = 2.0$: $\max(3, 2) = 3$, *so* 21 000; $w = 3.2$: 28 000 — *the rows of the test-case table.* `PerItemRule` *and* `FlatRule` *are almost trivial, and that is a virtue. Book: Section 6.16.3.*

Rules That Wrap Other Rules

Express and member hold another `PriceRule`, ask it, and adjust: the **decorator** pattern. Express is Equation (6.6) (Listing 6.24), member is Equation (6.7).

$$P_{\text{express}} = P + \lfloor \frac{P}{2} \rfloor \tag{6.6}$$

$$P_{\text{member}} = P - \lfloor \frac{p \cdot P}{100} \rfloor \tag{6.7}$$

Listing 6.24 The method `priceOf` of `ExpressRule.java`: a decorator that adds 50%, rounding down.

```
    @Override
    public long priceOf(Order order) {
        long normal = base.priceOf(order);
        return normal + normal / 2;          // integer division: half a rupiah is dropped
    }
```

Odd $P = 21\,001$*:* $21\,001 + 10\,500 = 31\,501$*. Member: write* `normal * percent / 100`*;* `normal * (percent / 100)` *computes* $10/100 = 0$ *in integers and gives no discount at all. Book: Section 6.16.4.*

Worked Example 4: Budi's Chain

3.2 kg of wash and fold, express, member: `new MemberDiscountRule(new ExpressRule(new PerKiloRule("wash & fold", 7_000, 3)), 10)`.

$$7\,000 \cdot \max\left(3, \lceil 3.2 \rceil\right) = 28\,000 \xrightarrow{\text{express}} 42\,000 \xrightarrow{\text{member}} 37\,800 \tag{6.8}$$

Table 6.5 The five price rules of Profitina Laundry.

Rule	Price	Refuses
`PerKiloRule(name, r, m)`	Equation (6.5)	an order with no weight
`PerItemRule(name, price)`	price × pieces	an order with no pieces
`FlatRule(name, amount)`	amount	nothing
`ExpressRule(base)`	Equation (6.6) on the base price	whatever the base refuses
`MemberDiscountRule(base, p)`	Equation (6.7) for members, base price otherwise	p outside $0..100$ (when built)

From the inside out (Equation (6.8)): $7\,000 \cdot 4$, *then* $+\,14\,000$, *then* $-\,4\,200$. *Table 6.5 gives every rule's price and what it refuses. Book: Section 6.16.4.*

The Pricing Design

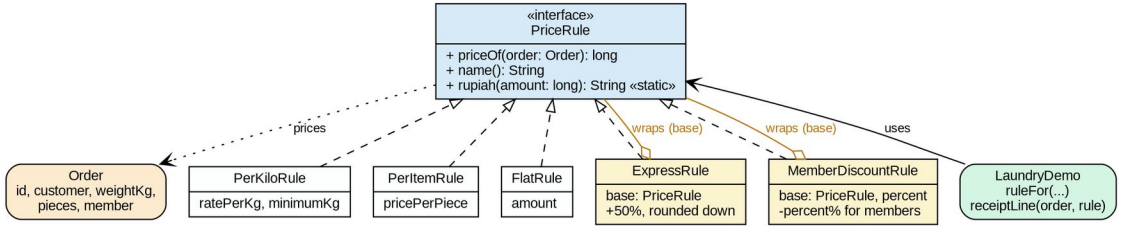


Figure 6.7 The pricing design: five classes implement `PriceRule`, two of them by wrapping another `PriceRule`; the demo program depends only on the interface.

Figure 6.7: five classes implement `PriceRule`, two by wrapping another `PriceRule`; the demo depends only on the interface. Book: Section 6.16.4.

A Morning's Receipts

`ruleFor` is a small *factory*, the only code that names the rule classes; `receiptLine` knows only the interface (Listing 6.25).

Listing 6.25 The method `receiptLine` of `LaundryDemo.java`: it works for every rule that exists and every rule written tomorrow.

```
    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)));
    }
```

```
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
```

A002: $40\,000 \to 60\,000 \to 54\,000$. *A006 (dry cleaning with no pieces) is refused, and the refusal passes unchanged through the member wrapper. Total* $263\,000$. *Book: Section 6.16.5.*

Testing Every Rule

Every class of the test-case table, both sides of every boundary, every refusal, the rounding with an **odd** price, and the stack (Listing 6.26).

Listing 6.26 From `test_price_rules.py`: a parametrized boundary test and a test with a stand-in rule.

```
@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_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
```

`@pytest.mark.parametrize` runs one body once per row. `FlatRule("odd", 21_001)` is a stub: through the interface, `ExpressRule` cannot tell it from a real rule, so express is tested alone. Book: Section 6.16.6.

What the Interface Bought Us

A rainy-season promotion, 5% off in January: one new class `SeasonalRule` wrapping a base rule, one line in `ruleFor`, a handful of tests.

`Order`, `PriceRule`, the five rules, `receiptLine` and every existing test stay untouched — and the old tests keep proving the old prices.

The *documentation* says what every rule promises, the *tests* check each rule keeps it, the *interface* lets the program rely on it.

That is the whole chapter in one page. Exercise 6.11 adds a bulk discount of this kind. Book: Section 6.16.7.

Profitina in Practice

One interface, many signals: every readiness signal is a class behind "given a scanned site, produce a score and an explanation"; a new signal is a new class.

A stand-in for tests, never for users: code that needs the local model (Qwen2.5 on Ollama) uses a narrow interface; tests plug in a fixed answer, production says plainly when no model is available.

Tests as a release gate: well over a thousand tests must be green, and a release counts only when the new version passes its health check.

Equality: `https://Example.com/` and `https://example.com` must be one page — `Customer`'s normalized phone number in another costume.

FastAPI builds API pages from annotations and docstrings — `pydoc`'s idea, grown up. Book: Section 6.17.

Common Mistakes

Documentation: a comment ended with `/`; `@param` names that no longer match; describing the implementation instead of the contract.

Tests: recomputing the answer with the same formula; tests that depend on each other; only the pleasant cases; `==` on doubles; expected and actual swapped.

Equality: `equals` without `hashCode`; `equals(Customer other)`; `==` on strings; the default `toString`, or secrets in it.

Overloading and interfaces: one name for different meanings; assuming `compareTo` returns ± 1; declaring `ArrayList` or `TreeDB` where `List` or `DBops` would do; rounding money in the wrong place.

Fourteen mistakes are listed in the book, each with its fix. Book: Section 6.18.

Summary: Five Things to Remember

1. Document every public item with Javadoc or a docstring; let `javadoc`, `pydoc` and `doctest` build and check it.

2. A unit test is arrange–act–assert against a literal worked out beforehand; choose inputs by classes and boundaries; compare doubles with a tolerance (Equation (6.1)).

3. Identity is not equality: override `equals` and `hashCode` together, from the same fields (Equation (6.2)).

4. Overloads are chosen at compile time by exact match, widening, boxing; Python uses defaults, `*args` and `singledispatch`.

5. An interface is a promise without bodies; program to it, and new behavior becomes a new class.

Profitina Laundry's prices are five classes behind `PriceRule`, each tested alone. Book: Section 6.19.

Next Time

Chapter 7 — Inheritance, Polymorphism, and Access Control. Classes that extend classes, overriding, dynamic dispatch explained, abstract classes, and who may see what.

Today's `Measurable` and `DBops` become class hierarchies with shared state.

Before then: work Exercises 6.1, 6.4, 6.7 and 6.11, in Java and in Python.

Dynamic dispatch, which today made `average` work for every `Measurable`, is the heart of Chapter 7. Book: Sections 6.20 and 6.21.

Lecture 7 — Inheritance, Polymorphism, and Access Control

Course: Object-Oriented Programming (EF234302)

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*, Chapter 7

Lecturer: Irfan Subakti — yifana@gmail.com

Today one line, `s.price()`, learns to price washing, ironing, dry cleaning and express orders without asking which is which. Book: Chapter 7, Sections 7.1–7.18.

What This Meeting Covers

1. **Inheritance** — the duplication problem, superclasses and subclasses, overriding, constructors and `super`.
2. **Polymorphism** — one call, many behaviors; static and dynamic types; method tables; Python's MRO.
3. **Designing hierarchies** — abstract classes, concrete superclasses, composition, the substitution principle.
4. **Multiple inheritance** — Java's interfaces and conflicting defaults; Python's diamond and C3.
5. **Access control and encapsulation** — Java's four levels, Python's conventions, invariants, properties.
6. **Profitina Laundry** — a service hierarchy priced by one polymorphic loop.

The order is the one a working programmer meets: a problem, its cure, the surprises the cure brings, and the discipline that keeps a hierarchy healthy. Book: Section 7.1.

Where We Left Off: a Class Full of Special Cases

Chapter 3's `Service` answered "how long?" with a `switch` on its own code, and priced express with an `if`. Chapter 6 gave us interfaces and overloading.

Add dry cleaning (per piece), shoes (per pair), a 20% promotion: each means reopening `Service`, another `case`, another `if`. After a year, a thousand lines nobody dares to touch.

> **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*.

Everything in this chapter is a detail of one of these three sentences. Book: Section 7.1.

Two Databases That Repeat Each Other

`LinearDB` keeps integer keys in an array; `TreeDB` in a binary search tree. Both 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?
```

`addSeveral` calls `addKey` once per key; `findOneOf` calls `search` until one succeeds. Neither cares **how** keys are stored.

Written as two unrelated classes, those two methods exist twice, letter for letter — and the day a bug is fixed in one copy, the two classes quietly disagree. Book: Section 7.2.

The Problem and the Cure

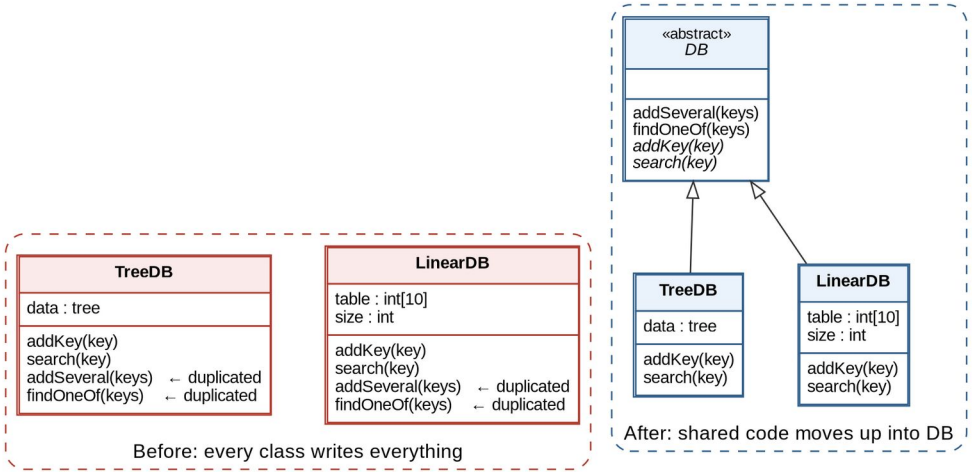


Figure 7.1 The duplication problem and its cure. Left: two independent classes each carry their own copy of `addSeveral` and `findOneOf`. Right: the shared methods move up into the abstract superclass `DB`; each subclass keeps only what is genuinely its own.

Figure 7.1: the shared methods move up into `DB`, which also declares the two whose code differs. The idea goes back to Simula 67. Book: Section 7.2.

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.

Say it aloud: **is-a**. A `LinearDB` *is a* `DB`. If the sentence sounds wrong — "a `Customer` is a `Service`" — the inheritance is wrong, however much code it would save.

Inheritance is **transitive**: if `D extends C` and `C extends B`, then `D` has everything of `C` and of `B`.

A useful mental model: the subclass behaves as if the superclass's members had been copied into it. No code is really copied; Section 7.7 shows what happens. Book: Sections 7.2.1 and 7.3.1.

The Abstract Superclass `DB`

Listing 7.1: `addSeveral` has an ordinary body that calls `addKey` — which `DB` only *declares*.

Listing 7.1 From `db/DBTest.java`: the abstract class `DB`, with one complete method and two abstract ones.

```
abstract class DB {
    void addSeveral(int keys[]) {
        for (int i = 0; i < keys.length; i++) {
            addKey(keys[i]);
        }
    }
    abstract void addKey(int key);     // No code
    abstract boolean search(int key);  // No code
}
```

`abstract` means "incomplete: you may not create objects of it". `findOneOf` is written the same way, a loop that calls `search`. Book: Section 7.3.

A Subclass: `LinearDB extends DB`

Listing 7.2 adds two fields and supplies the two missing bodies; here they only report that they ran.

Listing 7.2 From `db/DBTest.java`: the subclass `LinearDB`.

```
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;
    }
}
```

`DB ldb = new LinearDB();` — variable of type `DB`, object of class `LinearDB`. `ldb.addSeveral(new int[] {3, 4})` runs `DB`'s loop, which prints `LinearDB: addKey()` twice: code in the superclass calls methods chosen by the object. Book: Section 7.3.

The Same Project in Python

Listing 7.3: `class LinearDB(DB):` — no `extends`; `ABC` and `@abstractmethod` give Python the "incomplete class".

Listing 7.3 From `db_test.py`: `DB` inherits from `ABC` and marks `add_key` and `search` as abstract.

```
class DB(ABC):
    def add_several(self, keys):
        for key in keys:
            self.add_key(key)
    @abstractmethod
    def add_key(self, key):  # No code
        ...

    @abstractmethod
    def search(self, key):  # No code
        ...
```

```
DB(): Can't instantiate abstract class DB with abstract methods add_key, search
```

Java refuses `new DB()` at compile time ("DB is abstract; cannot be instantiated"); Python refuses `DB()` at run time, with the `TypeError` above. Fields are created by assigning `self.table` in `__init__`. Book: Section 7.3.2.

Overriding: Redefining an Inherited Method

> **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.

Listing 7.4: an array database can add many keys with one bulk copy.

Listing 7.4 From `IntDBDemo.java`: `LinearIntDB` overrides the inherited `addSeveral`; `TreeIntDB` inherits it unchanged.

```
    // 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)");
    }
```

*Always write `@Override`: it turns a misspelled name or a wrong parameter type into a compile error. In Python a method overrides if it has the same **name**; the parameters do not matter. Book: Section 7.4.*

Worked Example 1: Counting Comparisons

Both databases get $\langle 42, 7, 19, 3, 25 \rangle$ and the same searches; both count comparisons:

```
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
```

By hand, the array: `search(19)` looks at 42, 7, 19: 3. `search(8)`: 5. `findOneOf({8,9,25})`: 5+5+5=15. `findOneOf({1,2})`: 10. Total 3+5+15+10=33.

`(bulk copy of 5 keys)` *shows the override ran. The tree answers the same, in 22 comparisons: every search follows one root-to-leaf path of at most four nodes (Exercise 7.2). Book: Section 7.4.*

The Rules Java Checks When You Override

1. **Return type:** the same, or a subtype (*covariant*): `Garment make()` may become `Shirt make()`.
2. **Access:** never more restrictive — a `public` method cannot be overridden by a package-private one ("attempting to assign weaker access privileges").
3. **`final`:** a `final` method cannot be overridden; a `final` class (like `String`) cannot be extended.
4. **`static` and `private`** methods are not overridden: a subclass method *hides* the first and is unrelated to the second.
5. **Checked exceptions** may shrink, never grow.

Every rule protects a promise the superclass made to its callers. Python enforces none of them; its mistakes surface only when the program runs. Book: Section 7.4.1.

Constructors Chain Upward

The superclass part is built **first**, by the superclass's own constructor. Listing 7.5: a `Shirt` is a `Garment` is a `LaundryItem`.

Listing 7.5 The class `Shirt` in `ConstructorChain.java`: `super(...)` first, and `super.describe()` to extend the parent's text.

```
    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
```

We asked for a `Shirt`, yet `LaundryItem` speaks first. `super.describe()` means "do what my parent does, then a little more": `item #1`, `, a blue shirt`, `, starched`. Book: Section 7.5.

The Chain, Drawn

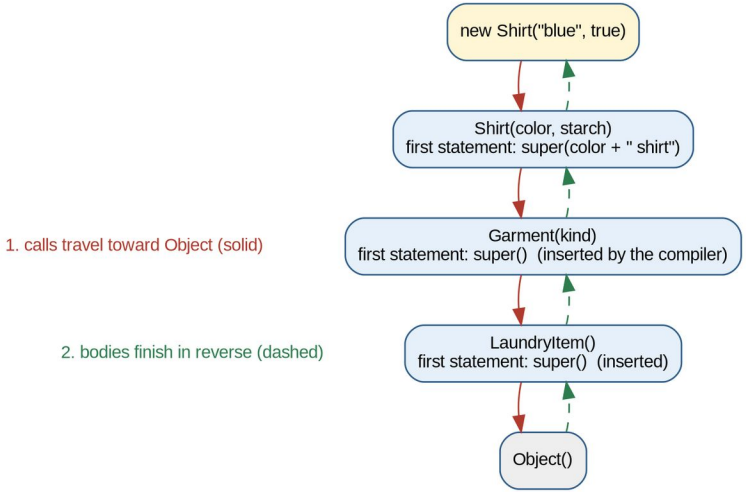


Figure 7.2 Constructor chaining for `new Shirt("blue", true)`. The calls travel up the hierarchy to `Object` first (solid arrows); the constructor bodies then finish in reverse order (dashed arrows), so the most general part of the object is always ready before the most specific part.

Figure 7.2: `Garment`'s constructor begins with an invisible `super();` inserted by the compiler, and so does `LaundryItem`'s, up to `Object`; then the bodies finish in reverse order. Book: Section 7.5.

Three Rules of Construction

1. **`super(...)` comes first**, if it appears at all: nothing may happen to a `Shirt` before its `Garment` part exists.
2. **No `super(...)`? The compiler writes `super()`** — and it must be legal. Delete `super(color + " shirt")` and `javac` reports that `Garment` "cannot be applied to given types; required: String, found: no arguments".
3. **Constructors are not inherited.** Each class states how its own objects are built.

> **PYTHON NEVER INSERTS THE CALL.** In Listing 7.6, `ForgetfulGarment` never runs `LaundryItem.__init__`, so its object has no `tag`:

Listing 7.6 From `constructor_chain.py`: a subclass that forgets `super().__init__()`.

```
class ForgetfulGarment(LaundryItem):
    def __init__(self, kind):
        self.kind = kind  # forgot super().__init__(): no tag is ever set
```

```
-- ForgetfulGarment("sock") --
  AttributeError: 'ForgetfulGarment' object has no attribute 'tag'
```

The error appears far from its cause. The habit: the first line of every subclass `__init__` is `super().__init__(...)`. Book: Section 7.5.

Polymorphism: the `Animal` Program

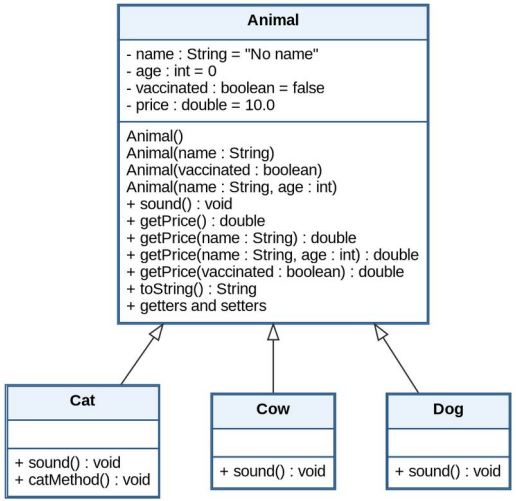


Figure 7.3 UML class diagram of the Animal program. `Animal` has four overloaded constructors and four overloaded `getPrice` methods; `Cat`, `Cow` and `Dog` each override `sound()`, and `Cat` adds a method of its own. A hollow triangle points from a subclass to its superclass; `-` marks a private member and `+` a public one.

Figure 7.3: `Animal` has four overloaded constructors and four overloaded `getPrice` methods; `Cat`, `Cow` and `Dog` override `sound()`, and `Cat` adds `catMethod()`. Book: Section 7.6.

One Call, Many Behaviors

In Listing 7.7 all four variables are declared `Animal`. The four calls look identical:

Listing 7.7 From `animal/AnimalTest.java`: four identical calls, and the `Cat` that overrides `sound()`.

```
        // Polymorphism
        animal.sound();
        cat.sound();
        cow.sound();
        dog.sound();
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");
    }
```

```
Animal makes a sound
Cat says: meow
Cow says: moo
Dog says: woof
```

*The compiler could not choose, because it knows only that each variable is an `Animal`; the running program looked at the object. That is **run-time polymorphism**. Book: Sections 7.6.1 and 7.6.2.*

What the Compiler Allows

`Animal c1 = new Cat();` — the object has `catMethod()`, but `c1.catMethod()` does not compile:

```
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
```

The compiler reasons from the declared type. `((Cat) c1).catMethod()` works — checked when the program runs; `((Cat) a1)` on a plain `Animal` throws `ClassCastException`.

Python has nothing to cast: `c1.cat_method()` simply works, and `a1.cat_method()` fails only at the call, with `AttributeError` — duck typing. Book: Sections 7.6.2 and 7.6.3.

Overloading versus Overriding

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

Table 7.1: the compiler chooses an overload; the running object chooses an override. Python replaces overloading with default and keyword arguments: `Animal(vaccinated=True)`. Book: Section 7.6.3.

Two Types for Every Variable

> **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.

`Machine m = new Washer("W-01");` is an **upcast**, always safe. The **compiler** uses the static type to decide whether a call is *allowed*; the **running program** uses the dynamic type to decide *which override* runs.

*This division of labor is **dynamic dispatch** (late binding). It applies only to instance methods that can be overridden. Book: Section 7.7.1.*

Worked Example 2: Three Surprises

Listing 7.8: predict each line before reading it.

Listing 7.8 From `DispatchDemo.java`: a method, a field and a static method, all reached through `m`.

```
        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());
```

```
W-01: fill water, spin at 1200 rpm
m.label  = machine
m.kind() = Machine.kind()
m.describe() = W-01 is a machine
```

1. `start()` is dispatched: the washer's line. **2.** `Washer` redeclares `label`: it *hides* `Machine.label`, and the static type picks `machine`. **3.** `kind()` is `static`: bound to `Machine`.

Never redeclare a superclass field; call static methods through the class name. `x instanceof Dryer d` *tests and casts in one step;* `getClass()` *reports the dynamic type. Book: Section 7.7.1.*

What Is Decided When

Table 7.2 What is decided when: only overridable instance methods are dispatched on the dynamic type.

Member	Chosen by	At	In Listing 7.8
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

Table 7.2: only overridable instance methods are dispatched on the dynamic type; fields, `static` and `private` methods, and the choice among overloads are all fixed by the compiler. Book: Section 7.7.1.

The Memory Picture: Method Tables

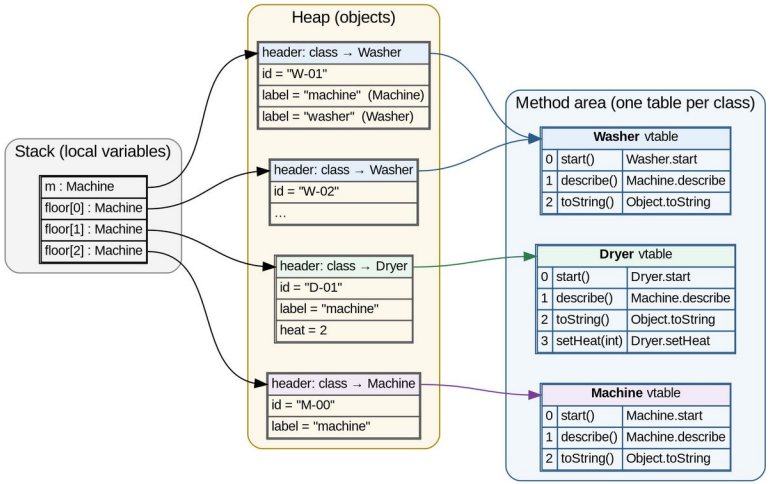


Figure 7.4 Memory while Listing 7.8 runs. Local variables on the stack hold references to objects on the heap; every object's header points to its class; every class has one method table (*vtable*) whose slot k holds the code for the k-th overridable method. Subclasses copy their parent's slots and overwrite the ones they override.

Figure 7.4: stack references, heap objects whose header points to their class, and one vtable per class. `Washer` *copies* `Machine`*'s slots and overwrites slot 0;* `Dryer` *also appends* `setHeat`*. Book: Section 7.7.2.*

A Call Is Three Reads, Not a Search

Each overridable method gets a fixed **slot**: `start()` 0, `describe()` 1, `toString()` 2. The call `x.start()` becomes Equation (7.1):

$$\text{code to run} = \text{vtable}(\text{class}(x))[\text{slot}(\text{start})] \tag{7.1}$$

Follow `x` to the object; the header to its class's vtable; jump to the code in the slot the compiler fixed. Two classes or two hundred: the same few reads.

The picture explains the rules: a field sits at a fixed offset chosen by the compiler; static and private methods never enter a vtable.

The JVM specification prescribes the behavior of `invokevirtual`, not this data structure; real JVMs use vtables and even call a method directly when a call site only ever meets one class. Book: Section 7.7.2.

Python: a Walk Along the MRO

Python checks the object's own dictionary, then each class of `type(m).__mro__` in order (Listing 7.9).

Listing 7.9 The helper `where_is` from `dispatch_demo.py`: the lookup Python performs, in six lines.

```
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
```

```
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
```

> **KEY INSIGHT.** Java decides "may I call this?" from the *static* type and "which version?" from the *dynamic* type; Python decides both at run time from the object. Either way, **the object decides which method runs**.

In Python `label` is a class attribute found by the same walk, so `m.label` prints `washer`: no hidden-field trap. Every class descends from `Object` / `object`, whose `toString`, `equals` and `hashCode` are meant to be overridden. Book: Sections 7.7.3 and 7.8.

Abstract Classes

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 may hold fields, constructors, complete methods and abstract methods at once: **what a family shares, including code**, with the differing parts left to subclasses.

An abstract class sits between an interface, which has only method declarations, and a concrete class, which has only complete methods. Book: Section 7.9.

Hierarchies with Abstract Levels

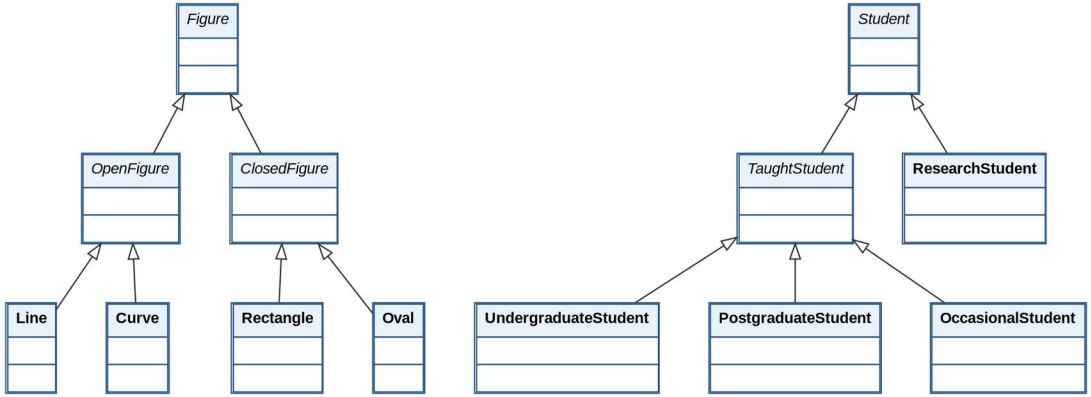


Figure 7.5 Two hierarchies with abstract inner levels (names in italics). Left: geometric figures, split into open and closed figures. Right: students, split into taught and research students.

Figure 7.5: figures split into open and closed ones; students into taught and research students (Exercise 7.4). Abstract names are in italics. Book: Section 7.9.

Two Formulas for an Oval

Semi-axes a, b: the area is Equation (7.2); the perimeter has no elementary formula, so we use Ramanujan's approximation Equation (7.3).

$$A = \pi\, a\, b \tag{7.2}$$

$$P \approx \pi\left[3(a+b) - \sqrt{(3a+b)(a+3b)}\right] \tag{7.3}$$

Listing 7.10 The two methods of `Oval` in `FigureDemo.java`.

```
    @Override
    double length() {  // Ramanujan's approximation of the perimeter
        return Math.PI * (3 * (a + b) - Math.sqrt((3 * a + b) * (a + 3 * b)));
    }

    @Override
    double area() {
        return Math.PI * a * b;
    }
```

Listing 7.10 is the two formulas line for line. Equation (7.3) is exact for a circle: $a = b = r$ *gives* $\pi\left[6r - \sqrt{16r^2}\right] = 2\pi r$. *Book: Section 7.9.*

Worked Example 3: Check the Figures

`Figure.describe()` is a **template method**: complete, but it calls the abstract `length()`. `ClosedFigure` adds `area()` and extends `describe()` with `super.describe()`.

```
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
```

Line $(0,0)$–$(3,4)$: $\sqrt{3^2+4^2}=5$. Curve: $2\sqrt{2}\approx 2.828$. Rectangle: $2(4+2.5)=13$, area 10.

Oval 3×2: area $6\pi\approx 18.850$; perimeter $\pi\left(15-\sqrt{99}\right)\approx 15.865$. Closed areas: $10+18.850=28.850$.

`Locale.US` keeps the decimal point on every computer; `new Figure("?")` does not compile. Book: Section 7.9.

The Java Library Is Built the Same Way

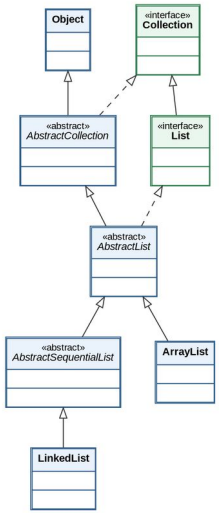


Figure 7.6 Part of the Java collections hierarchy. Solid lines with hollow triangles are `extends`; dashed lines are `implements`. The abstract classes supply shared code, the interfaces state the contracts, and `ArrayList` and `LinkedList` are the concrete classes programs actually create.

Figure 7.6: `AbstractList` implements `contains`, `indexOf`, `iterator` and more from just `get(int)` and `size()`. A new kind of list writes two methods and inherits dozens. Book: Section 7.9.1.

Inheriting from a Concrete Class

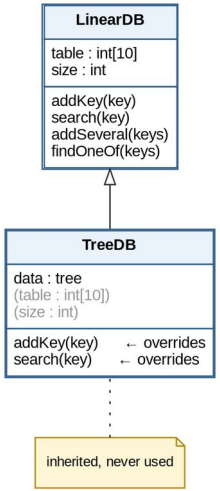


Figure 7.7 Inheriting from a concrete class. `TreeDB` receives `LinearDB`'s array and counter although a tree never uses them, and every `TreeDB` now claims to be a `LinearDB`.

*Figure 7.7: `TreeDB extends LinearDB` works, but carries an unused array (**baggage**), lets `LinearDB x = new TreeDB()` compile (**a false promise**), and depends on `LinearDB`'s insides (**coupling**). Extract an abstract superclass instead. Book: Section 7.10.*

Composition: Has-a Instead of Is-a

> **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.

Favor object composition over class inheritance. A subclass depends not only on what its superclass *promises* but on *how it is implemented*.

An `Order` has services; a car has an engine. Inheritance creates the tightest coupling there is between two classes. Book: Section 7.11.

Worked Example 4: the Fragile Base Class

Listing 7.11: a tag set that counts every tag ever added, by extending `HashSet<String>`. Three tags go in.

Listing 7.11 From `FragileBase.java`: the two overrides of `InheritedTagSet`.

```
    @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!
    }
```

```
inherited: size = 3, tags added = 6
composed:  size = 3, tags added = 3
```

`addAll` adds 3, then `super.addAll` calls `add` three times — and dispatch sends those calls back to our `add`: six. The composed version holds a `Set` and counts 3. Book: Section 7.11.1.

Why the Count Doubles

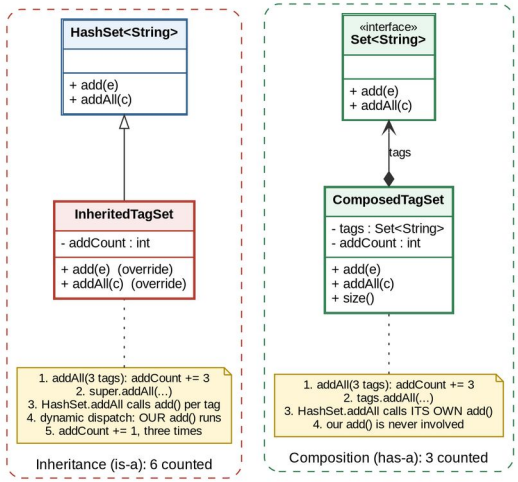


Figure 7.8 The fragile base class. Left: the subclass's `addAll` calls the superclass's `addAll`, which calls `add`, which dynamic dispatch sends back to the subclass: every tag is counted twice. Right: the composed class calls methods of an object it merely holds; its own `add` is never re-entered.

Figure 7.8: whether `addAll` calls `add` is an undocumented detail of `HashSet`; a correct subclass can break when it changes. Python's `UserDict` springs the same trap. Book: Section 7.11.1.

When Is Inheritance Right?

The **Liskov substitution principle**, Equation (7.4): whatever can be proved of every T must hold for every subtype S.

$$\forall\, x : T \,.\, \phi(x) \Rightarrow \forall\, y : S \,.\, \phi(y) \text{ whenever } S \text{ is a subtype of } T \tag{7.4}$$

1. Does the is-a sentence sound right to a non-programmer? **2.** Would every caller of the superclass be happy with a subclass object? **3.** Do you want the subclass to change when the superclass's insides change?

The laundry does both: `ExpressService` *is a* `Service` and *has a* `Service` — the **decorator** pattern.

If callers rely on a positive price, no subclass may return zero; if they rely on insertion order, a `TreeDB` *may not pose as a linear database. Book: Section 7.11.2.*

Java: One Superclass, Many Interfaces

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

Table 7.3: the danger of multiple inheritance is inheriting state and code along two paths; interfaces carry no state. Book: Section 7.12.1.

Two Defaults with One Signature

`Foldable` and `Steamable` each have a default `finish()`. `PremiumCare` implements both and **must** choose (Listing 7.12):

Listing 7.12 From `DiamondDemo.java`: resolving two conflicting defaults with `Interface.super.method()`.

```
// 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();
    }
}
```

```
PremiumCare: steamed, then folded
as Foldable:  steamed, then folded
as Steamable: steamed, then folded
HotelCare:   hung on a rack
```

Without the override, `javac` stops: "inherits unrelated defaults for finish()". Rules: a class method beats a default; a more specific interface beats a less specific one; otherwise the class must choose. Book: Section 7.12.1.

Python's Diamond

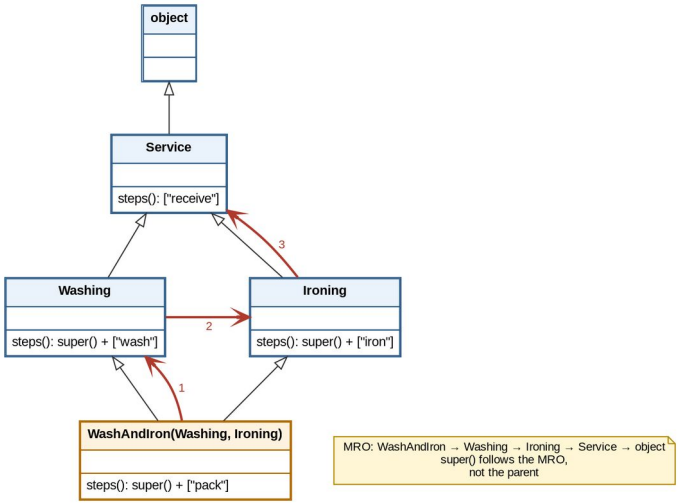


Figure 7.9 The diamond. `WashAndIron` has two paths up to `Service`. Python linearizes the hierarchy into one method resolution order, and every `super()` call moves to the next class in that order (numbered arrows), so `Service.steps` runs exactly once.

Figure 7.9: `WashAndIron` has two paths to `Service`; Python linearizes them into one MRO, and each `super()` moves to the next class in it, so `Service.steps` runs once. Book: Section 7.12.2.

`super()` Is Not "My Parent"

Listing 7.13: every `steps` calls `super().steps()` and adds one word.

Listing 7.13 From `diamond_mro.py`: a diamond with cooperative `super()` calls.

```
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"]
```

```
WashAndIron MRO: ['WashAndIron', 'Washing', 'Ironing', 'Service', 'object']
steps(): ['receive', 'iron', 'wash', 'pack']
```

*`Washing`'s `super()` goes to `Ironing` — not its superclass, but the next class in `WashAndIron`'s MRO. That is **cooperative multiple inheritance**. Book: Section 7.12.2.*

Worked Example 5: C3 by Hand

The linearization Equation (7.5): take the first head that appears in no other list's tail.

$$L[C] = C + \text{merge}\big(L[B_1], \ldots, L[B_n], [B_1, \ldots, B_n]\big) \tag{7.5}$$

Merge $[W, S, o]$, $[I, S, o]$, $[W, I]$: take W. Head S is in a tail — skip; take I. Now take S, then o.

$L[\text{WashAndIron}] = [\text{WashAndIron}, W, I, S, o]$, with W = `Washing`, I = `Ironing`, S = `Service`, o = `object`.

A shared ancestor waits until all its descendants are listed. `class Confused(Service, Washing)` *has no consistent order, and Python raises* `TypeError`*. Book: Section 7.12.2.*

Java's Four Access Levels

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

A top-level class is `public` or package-private. Members take one of four levels: `public`, `protected`, none (package-private), `private`.

Table 7.4 is the official reference table: "Y" means code in that place can access a member declared with that modifier. Book: Section 7.13.1.

Four Classes, Two Packages

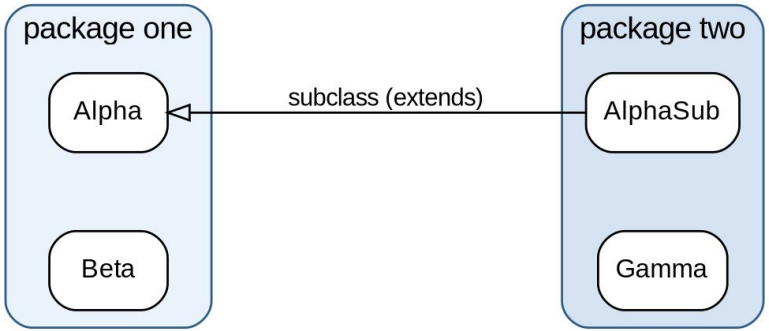


Figure 7.10 Four classes in two packages. `Beta` shares a package with `Alpha`; `AlphaSub` is in another package but extends `Alpha`; `Gamma` has no relationship with `Alpha` at all.

Figure 7.10: `Beta` shares a package with `Alpha`; `AlphaSub` is elsewhere but extends `Alpha`; `Gamma` has no relationship with it. Book: Section 7.13.1.

Worked Example 6: Let the Compiler Prove It

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

`java -cp out two.Gamma` prints one column of Table 7.5 per class; adding `a.protectedField`, `a.packageField`, `a.privateField` to `Gamma` gives three different errors.

The subtlety: `AlphaSub` may use `protectedField` of **its own** object, but not `other.protectedField` of an arbitrary `Alpha`.

Use the most restrictive level that makes sense — `private` unless there is a good reason — and no public fields except constants. `protected` is a promise to every subclass anyone will ever write. Book: Sections 7.13.1 and 7.13.2.

Python: Conventions Instead of Rules

`name` public; `_name` internal ("subclasses welcome"); `__name` **mangled** to `_Alpha__name`, to avoid clashes, not for secrecy (Listing 7.14).

Listing 7.14 From `access_demo.py`: Python's three naming conventions.

```python
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}"
```

```
AlphaSub sees: public protected  (AttributeError: 'AlphaSub' object has no attribute
'_AlphaSub__private_field')
attribute names: ['public_field', '_protected_field', '_Alpha__private_field']
```

Inside `AlphaSub`, `self.__private_field` becomes `_AlphaSub__private_field`, which does not exist. "We are all consenting adults here": the boundary is marked, not enforced. Book: Section 7.13.3.

Encapsulation Keeps an Invariant

A washing machine promises Equation (7.6): load ℓ between 0 and capacity c. Only `addLoad` can raise the load (Listing 7.15).

$$0 \le \ell \le c \tag{7.6}$$

Listing 7.15 The method `addLoad` of `EncapsulationDemo.java`: the only way to raise the load, and it preserves the invariant.

```
    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;
    }
```

```
W-03: 5.5 / 8.0 kg
Refused: 3.0 kg does not fit, only 2.5 kg free
```

`m.loadKg = 99` does not compile: "loadKg has private access". And because nobody sees `loadKg`, it could later be stored in grams without changing another class — Parnas's information hiding. Book: Section 7.14.

Python Properties

A **property** is a method called like a field: Listing 7.16 runs the setter's check even for `m.load_kg = 99`.

Listing 7.16 From `encapsulation_demo.py`: `load_kg` as a property with a validating setter.

```
    @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
```

```
Refused: 99.0 kg is outside 0..8.0 kg
AttributeError: property 'capacity_kg' of 'WashingMachine' object has no setter
```

`capacity_kg` has no setter, so assigning to it raises `AttributeError`. Python programmers start with plain attributes and add properties when a rule appears; callers never notice. Book: Section 7.14.1.

Profitina Laundry: the Price Rules

Wash & Fold Rp7,000/kg, min 3 kg, 48 h; **Iron Only** Rp5,000/kg, min 2 kg, 24 h; **Wash, Dry & Iron** Rp10,000/kg, min 3 kg, 72 h; **Dry Clean** per piece, 72 h.

Chargeable weight Equation (7.7) and weight price Equation (7.8); express Equation (7.9); order total and member total Equation (7.10).

$$\hat{w} = \frac{\lceil 2 \max(w, w_{\min}) \rceil}{2} \tag{7.7}$$

$$P_{\text{weight}} = \text{round}(r \cdot \hat{w}) \tag{7.8}$$

$$P_{\text{express}} = \lfloor \frac{3P}{2} \rfloor \tag{7.9}$$

$$T = \sum_{i=1}^{n} P(s_i), T_{\text{member}} = T - \lfloor \frac{T}{10} \rfloor \tag{7.10}$$

4.2 kg of Wash & Fold: $\max(4.2, 3) = 4.2$*, doubled* 8.4*, up to* 9*, halved* 4.5 *kg,* $P = 31,500$*. Express is ready in a quarter of the time, never under 6 h. Book: Section 7.15.*

The Service Hierarchy

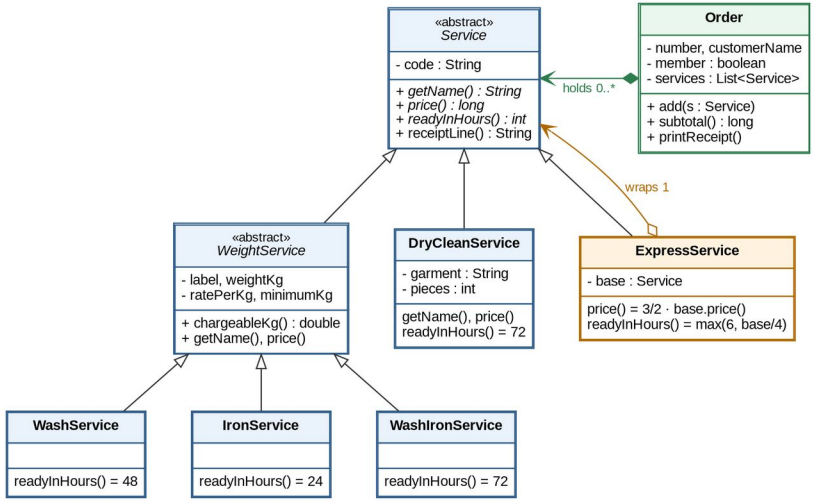


Figure 7.11 UML class diagram of the Profitina Laundry price engine. `Service` and `WeightService` are abstract; five concrete services extend them; `ExpressService` both *is* a `Service` and *wraps* one (hollow diamond); an `Order` holds any number of services (filled diamond).

Figure 7.11: `Service` and `WeightService` are abstract; five concrete services; `ExpressService` both is a `Service` and wraps one; an `Order` holds any number of services. Book: Section 7.15.

`Service`: Three Promises and a Template

Listing 7.17: every service has a code; each must say its name, price and turnaround; `receiptLine` formats all of them.

Listing 7.17 From `LaundryPricing.java`: the abstract methods of `Service` and its template method.

```
    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());
    }
```

`receiptLine` mentions no subclass. `Order.subtotal()` is a four-line loop, `sum += s.price()`, that mentions no subclass either — dynamic dispatch picks the right `price()`. Book: Section 7.15.

Almost-Empty Subclasses

`WeightService` holds the per-kilogram rules once; Listing 7.18 shows the chargeable weight and a whole concrete service.

Listing 7.18 From `LaundryPricing.java`: `chargeableKg` in `WeightService`, and `WashService` complete.

```
    // At least the minimum, rounded up to the next 0.5 kg.
    double chargeableKg() {
        return Math.ceil(Math.max(weightKg, minimumKg) * 2) / 2;
    }
class WashService extends WeightService {
    WashService(double weightKg) {
        super("WF", "Wash & Fold", weightKg, 7000, 3.0);
    }

    @Override
    int readyInHours() {
        return 48;
    }
}
```

Chapter 3's `switch` has become three one-line overrides of `readyInHours()`. Delete one, and `javac` refuses: "IronService is not abstract and does not override abstract method readyInHours()". Book: Section 7.15.

Express: Is-a and Has-a at Once

Listing 7.19: `ExpressService` extends `Service` **and** holds a `Service` called `base`.

Listing 7.19 From `LaundryPricing.java`: the decorator `ExpressService`.

```
    ExpressService(Service base) {
        super("EX-" + base.getCode());
        this.base = base;
    }
    @Override
    long price() {
        return base.price() * 3 / 2;  // 50% surcharge
    }

    @Override
    int readyInHours() {
        return Math.max(6, base.readyInHours() / 4);
    }
```

Because it accepts any `Service`, express works for dry cleaning, for services added next year, even for another express (Exercise 7.9). Book: Section 7.15.

Worked Example 7: Siti's Receipt

One order, six services, a member; one loop prices them all:

```
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
```

Iron 1.3 kg → 2 kg minimum, $10,000$. *Express WDI:* $\lfloor 3 \cdot 30,000/2 \rfloor = 45,000$, *ready in* $\max(6, 72/4) = 18$ *h. Subtotal* $236,500$; *discount* $23,650$; *total* $212,850$. *Python prints the same. Book: Section 7.15.*

Profitina in Practice

A Profitina scan examines many **signals** — clean extraction, schema.org structured data, whether `robots.txt` lets answer engines in.

One abstraction, *a signal*, promises: measure a page, give a score, explain it. Many concrete signal classes, each tested alone. The scan is one loop, like `Order.subtotal()`.

Shared behavior lives in a composed helper; personal data (UU PDP 27/2022) stays behind a few methods that enforce the rules, like `addLoad`.

The scores are advice: the platform explains what it measured; the user decides what to do. Book: Section 7.16.

Common Mistakes

Inheriting when is-a is false (`Order extends ArrayList<Service>`); **overloading when you meant to override** — `equals(Customer)`; write `@Override`.

Forgetting `super().__init__()` in Python; **redeclaring a superclass field** in Java; expecting **static** members to be polymorphic.

Downcasting without `instanceof`; `equals` without `hashCode`; `protected` **fields** "so subclasses can use them".

Reading `super()` as "my parent" in a Python diamond; **calling an overridable method from a constructor**, before the subclass's fields are set.

Ten mistakes are listed in the book, each with its fix. Book: Section 7.17.

Summary: Five Things to Remember

1. Inheritance moves shared code into a superclass; test it with "a C *is a* B". Constructors chain upward; Python calls `super().__init__()` only if you do.
2. Overriding: same name and parameters, chosen at run time; overloading: different parameters, chosen at compile time.
3. The static type decides what compiles; the dynamic type decides which override runs (Equation (7.1)); Python walks the MRO.
4. Prefer an abstract common superclass, and composition over inheritance unless substitution (Equation (7.4)) holds.
5. Use the most restrictive access that works; encapsulation keeps invariants; Python marks boundaries with `_`, `__` and `@property`.

In the laundry, one loop now prices every kind of service, and a new service is a new class, not a new `case` label. Book: Section 7.18.

Next Time

Chapter 8 — Evaluation and Discussion II. Worked problems on Chapters 5, 6 and 7: recursive trees and lists, a parser with exceptions, a growable array, and design problems.

Every problem is solved in full, in Java and in Python, and checked by a program.

Before then: work Exercises 7.2, 7.4, 7.9 and 7.10, in Java and in Python.

Bring today's hierarchies: next time we design one from a problem statement. Book: Sections 7.19 and 7.20.

Lecture 8 — Evaluation and Discussion II: Recursive Structures, Robust Classes, and Class Hierarchies

Course: Object-Oriented Programming (EF234302)

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*, Chapter 8

Lecturer: Irfan Subakti — yifana@gmail.com

No new language features today: we work problems, twenty-one of them, each solved from a blank page in Java and in Python and checked by a program. Book: Chapter 8, Sections 8.1–8.12.

What This Meeting Covers

1. **A method** — eight steps from a paragraph of requirements to classes, tests and code; the delegation recipe for recursion.
2. **Two structures** — a `Tree` that works by delegation, and an immutable list built from `nil` and `cons`.
3. **Problems 8.1–8.7** — building, tripling, traversing, copying and saving trees; search and insertion in a BST.
4. **Problems 8.8–8.15** — maps, searches, filters, sortedness, merging and duplicates on recursive lists.
5. **Problems 8.16–8.21** — a robust file parser, a growable array, an interface with contract tests, a hierarchy traced by hand, a refactoring, and a bug hunt.

Every problem goes the same route: the statement, in our own words, design, Java, Python, line by line, a tempting wrong solution, and a check-yourself variant. Book: Section 8.1.

Where We Stand: the Toolbox

Chapter 5: arrays and lists, input, exceptions (`try`/`catch`/`finally`, checked and unchecked, our own), and files closed safely.

Chapter 6: Javadoc and docstrings, JUnit 6 and `unittest`, interfaces, overloading.

Chapter 7: inheritance, abstract classes, overriding, polymorphism, `super`, `final`, and the access levels.

> **HOW TO USE TODAY.** Read a statement, cover the rest, and spend ten minutes on it yourself: draw the design, write the base case, try one example by hand. Then compare.

Each Java program tests itself with a small `check` helper that prints `PASS` or throws `AssertionError`; the Python programs use `unittest`. Book: Section 8.1.

The Eight-Step Method

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?

***Table 8.1**: Step 3, “who has the data this needs?”, prevents most bad designs; Step 7, tests with their edge cases, catches most wrong solutions. Book: Section 8.2.*

Responsibilities and the Delegation Recipe

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.

THE DELEGATION RECIPE. (1) *Base case*: the answer for the empty list or the missing subtree. (2) *Delegation*: what do I ask my smaller parts to compute? Assume they do it correctly. (3) *My own job*: how do I combine their answers with my own data?

If the three answers are right, the method is right, by induction on the size of the structure.

Prefer has-a and can-do relationships; use is-a only when the subclass really is a special case and really shares code. Book: Section 8.2.

A Tree That Works by Delegation

Listing 8.1: fields `private`; the leaf constructor chains to the branch constructor; `mirror()` asks each subtree to mirror itself, then swaps.

Listing 8.1 From `TreeStart.java`: the leaf constructor and `mirror()` by delegation.

```
        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;
        }
```

`toString()` writes a leaf as its number and a branch as `root(left,right)`, `-` for a missing subtree; every test compares that string. Python uses `_root` and default arguments. Book: Section 8.3.1.

The Tree of Problem 8.1

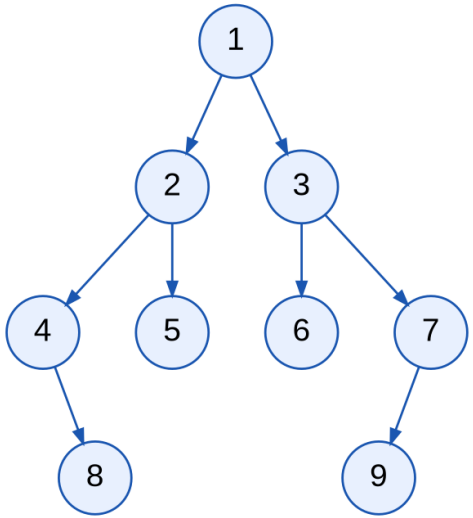


Figure 8.1 The binary tree of Problem 8.1. Node 4 has only a right child (8) and node 7 has only a left child (9); every other inner node has two children.

Figure 8.1 *prints as* `1(2(4(-,8),5),3(6,7(9,-)))`*: node 4 has only a right child and node 7 only a left one. Book: Section 8.3.1.*

An Immutable List from `nil` and `cons`

A list is either empty, `nil`, or `cons` of a *head* and a shorter *tail*, Equation (8.1):

$$[2, -5, 8] = \text{cons}(2, \text{cons}(-5, \text{cons}(8, \text{nil}))) \tag{8.1}$$

Immutable: `private final` fields; every "change" builds a new list, so sharing tails is safe. **Private constructors, public factories:** `nil()`, `cons()`, `of()`.

`ListException` for `head()` of the empty list, unchecked; `ListOps`, a utility class, not a superclass: `NegateAll extends List` would be a false is-a.

The names come from the Lisp family of languages. Python keeps the class and its helpers in one module, `cons_list.py`. Book: Section 8.3.2.

The Objects Behind `[2, -5, 8]`

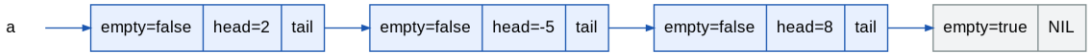


Figure 8.2 The objects behind `List a = cons(2, cons(-5, cons(8, nil())))`. Each cell has a head and a tail reference; the last tail is the single shared empty list.

***Figure 8.2**: three cells, each a head and a reference to its tail, and one shared empty list at the end. Book: Section 8.3.2.*

The List Classes in UML

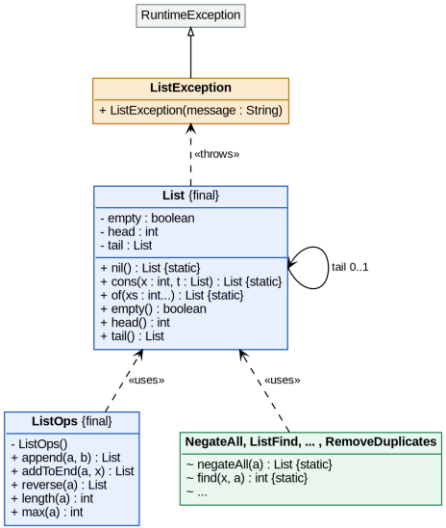


Figure 8.3 UML class diagram of the list classes. `ListException` is unchecked because it extends `RuntimeException`; every problem class of Section 8.5 only uses `List`, it does not extend it.

***Figure 8.3**: `ListException` extends `RuntimeException`; the problem classes only use `List`, which is `final`. Book: Section 8.3.2.*

Factories and a Broken Contract

Listing 8.2: nobody outside `List` can call `new List(...)`; the empty list is created once and shared.

Listing 8.2 From `List.java`: two factories, and `head()` refusing the empty list.

```
    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);
    }
    public int head() {
        if (empty) {
            throw new ListException("head() of the empty list");
        }
        return head;
    }
```

`max` of the empty list throws `ListException`, and a `catch` in `main` prints its message; without the `try` the program would stop with a stack trace. Book: Section 8.3.2.

Problem 8.1 — Building a Tree by Hand

PROBLEM. Create the tree of the figure with nine `new Tree` statements.

A branch needs its subtrees as arguments, so the tree is built **bottom-up** (**Listing 8.3**):

Listing 8.3 The method `build()` of `BuildTree.java`: nine constructor calls, leaves first.

```
    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
    }
```

```
t       = 1(2(4(-,8),5),3(6,7(9,-)))
wrong t = 1(2(4(8,-),5),3(6,7(9,-)))
```

The wrong version puts every lone child on the left: `4(8,-)`. It still has nine nodes, so only a test on the exact shape catches it. For a tree, left and right are part of the data. Book: Section 8.4.1.

Problem 8.2 — Tripling Every Node

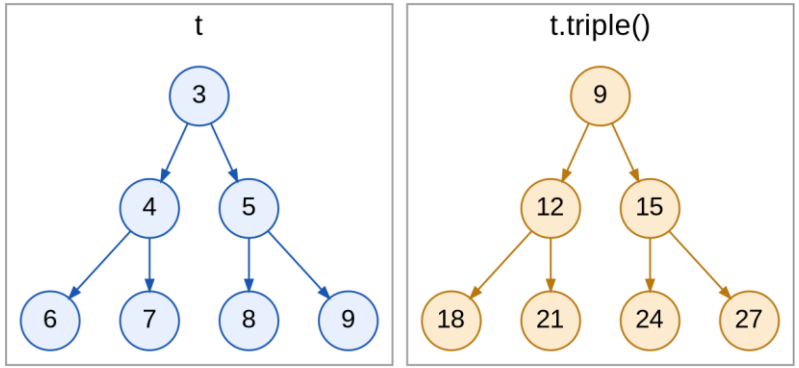


Figure 8.4 The sample tree before and after `t.triple()`: the shape is the same and every value is multiplied by three.

***Figure 8.4**: `void triple()` changes the tree in place; the shape stays, every value is multiplied by three. Book: Section 8.4.2.*

Problem 8.2 — Two Guarded Delegations

Listing 8.4: base case, the `null` guards; delegation, `left.triple()` and `right.triple()`; my own job, `root = 3 * root`.

Listing 8.4 The method `triple()` of `TripleTree.java`.

```
        void triple() {
            if (left != null) {
                left.triple();
            }
            if (right != null) {
                right.triple();
            }
            root = 3 * root;
        }
```

```
before: 3(4(6,7),5(8,9))
after : 9(12(18,21),15(24,27))
tripleWrong crashed: NullPointerException
half-done tree    : 9(12(18,7),5(8,9))
```

`tripleWrong()` drops the guards, "every node has two children": a leaf has none. The crash leaves a half-tripled tree — check before you change. Book: Section 8.4.2.

Problem 8.3 — The Depth-First Traversal

PROBLEM. Write `printDepthFirst()`, left branches first: for the sample tree it prints `3 4 6 7 5 8 9`.

Table 8.2 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)

***Table 8.2**: each row is one call; read the middle column top to bottom and you have the output. This is pre-order. Book: Section 8.4.3.*

Problem 8.3 — Where the Print Goes

Listing 8.5: print my root, then ask left, then right.

Listing 8.5 The method `printDepthFirst()` of `DepthFirstPrint.java`.

```
        void printDepthFirst() {
            System.out.print(root + " ");
            if (left != null) {
                left.printDepthFirst();
            }
            if (right != null) {
                right.printDepthFirst();
            }
        }
```

```
3 4 6 7 5 8 9
6 4 7 3 8 5 9  <- wrong order
```

Moving the `print` between the two calls gives in-order, `6 4 7 3 8 5 9`. The test captures `System.out` into a buffer and restores it in `finally`; Python uses `redirect_stdout`. Book: Section 8.4.3.

Problem 8.4 — Deep and Shallow Copies

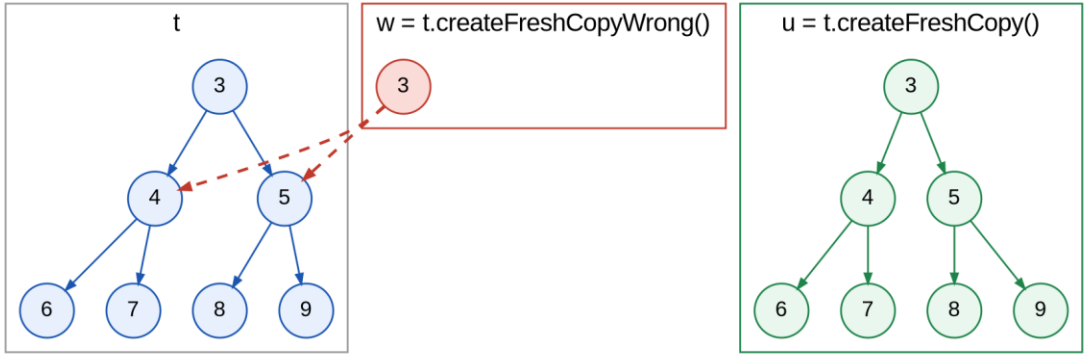


Figure 8.5 Left: the original `t`. Right: the fresh copy `u`, seven new objects. Middle: the wrong copy `w`, one new object whose references point back into `t`.

***Figure 8.5**: after `u = t.createFreshCopy(); t.mirror();`, `u` must not move. A deep copy is seven new objects; the wrong copy is one new root pointing back into `t`. Book: Section 8.4.4.*

Problem 8.4 — Keep What the Calls Return

Listing 8.6: a fresh copy of a node is a new node holding fresh copies of both subtrees.

Listing 8.6 The method `createFreshCopy()` of `FreshCopy.java`.

```
        Tree createFreshCopy() {
            Tree leftCopy = (left == null) ? null : left.createFreshCopy();
            Tree rightCopy = (right == null) ? null : right.createFreshCopy();
            return new Tree(root, leftCopy, rightCopy);
        }
```

```
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))
```

*`createFreshCopyWrong()` makes the recursive calls but **throws their results away** and builds `new Tree(root, left, right)`: a shallow copy, half right, which only the node-sharing test exposes. Book: Section 8.4.4.*

Problem 8.5 — Saving the Traversal into an Array

`saveDepthFirst(a, 4)` fills `? ? ? ? 4 5 7 8 6 9 10 ? ?` and returns the **last position written**, Equation (8.2):

$$end = s + n - 1, \tag{8.2}$$

Table 8.3 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

***Table 8.3**: the right subtree starts one cell after the last cell the left subtree used — a number only the left call knows, so it returns it.* $4 + 7 - 1 = 10$. *Book: Section 8.4.5.*

Problem 8.5 — One Convention, Written Down

Listing 8.7: `last` starts at `whereToStart`; each subtree starts at `last + 1` and its answer replaces `last`.

Listing 8.7 The method `saveDepthFirst` of `SaveDepthFirst.java`.

```
        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;
        }
```

```
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
```

The wrong version starts the right subtree at `last`, not after it: three values silently overwritten. Mixing "last used" with "next free" is the classic off-by-one. Book: Section 8.4.5.

Problem 8.6 — A Binary Search Tree

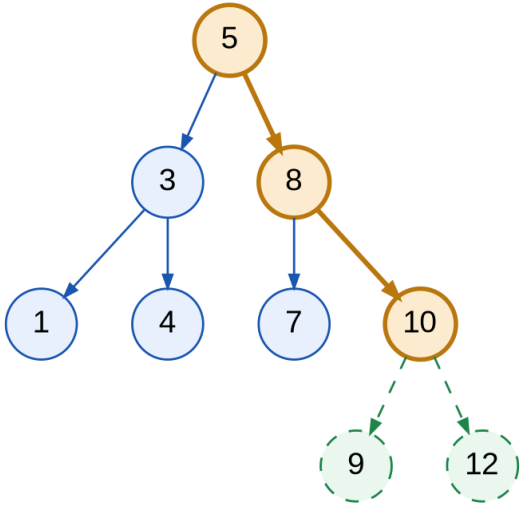


Figure 8.6 The binary search tree used in Problems 8.6 and 8.7. The thick path is followed by `find(8)` and `find(10)`; the dashed leaves 9 and 12 are added by `insert(9)` and `insert(12)`.

***Figure 8.6**: smaller values on the left, larger on the right; `find(8)` follows the thick path 5, 8 and stops. Book: Section 8.4.6.*

Problem 8.6 — One Path Down

Listing 8.8: one comparison decides which half cannot hold `n`; visits are bounded by the height, Equation (8.3).

$$\text{visits(find)} \le h, \qquad h \approx \log_2 n \text{ for a balanced tree,} \tag{8.3}$$

Listing 8.8 The method `find` of `BstFind.java`, with a visit counter.

```
        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);
            }
        }
```

```
find(8) = 8(7,10), nodes visited: 2
find(6) = null, nodes visited: 3
```

`findWrong` omits the `null` checks and crashes on `find(6)`. Searching both subtrees "to be safe" is right but visits every node; only a test that counts visits catches it. Book: Section 8.4.6.

Problem 8.7 — Insertion Is `find` with a New Ending

Listing 8.9: walk down as `find` does; where the needed subtree is missing, create the leaf **and attach it**.

Listing 8.9 From the method `insert` of `BstInsert.java` (the right-hand case is symmetric).

```
        Tree insert(int n) {
            if (n < root) {
                if (left == null) {
                    left = new Tree(n);          // attach the new leaf here
                } else {
                    left.insert(n);
                }
            }
            return this;                         // n == root: already there
        }
```

```
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)))
```

The in-order listing `1 3 4 5 7 8 9 10 12` proves the BST rule still holds. `insertWrong` creates the leaf but never stores it — an object no field refers to is garbage. Book: Section 8.4.7.

Recursive Lists: One Skeleton

Every list method below fills the two blanks of Equation (8.4):

$$f(a) = \begin{cases} \text{answer for the empty list} & \text{if } a = \text{nil}, \\ \text{combine}(\text{head}(a),\ f(\text{tail}(a))) & \text{otherwise.} \end{cases} \tag{8.4}$$

Only `empty()`, `head()`, `tail()`, `nil()` and `cons()` are allowed: no loops, no library lists. A list of n elements causes $n + 1$ calls.

Each method returns a new list; no list is ever changed. Book: Section 8.5.

Problem 8.8 — Negating a List

Listing 8.10 is a *map*: one output element per input element, `cons(-head, negateAll(tail))`.

Listing 8.10 The method `negateAll` of `NegateAll.java`.

```
    static List negateAll(List a) {
        if (a.empty()) {
            return List.nil();                          // base case
        }
        return List.cons(-a.head(), negateAll(a.tail()));  // recursive case
    }
```

```
a                = [2, -5, 8, 0]
negateAll(a)     = [-2, 5, -8, 0]
negateAllWrong(a) = [-2, -5, 8, 0]
```

The wrong version forgets to recurse and attaches the original tail. Negating twice must give the original list — a property test. In Java, $-(-2^{31})$ overflows back to -2^{31}. Book: Section 8.5.1.

Problem 8.9 — A Signal Passed Through

Listing 8.11: the recursive answer is either a position, shifted by one, or the signal -1, passed on **unchanged**.

Listing 8.11 The method `find` of `ListFind.java`.

```
    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;
    }
```

`findWrong` adds 1 to every answer: for an absent value in four elements, $-1 + 4 = 3$ — a valid position, the worst kind of wrong answer. Book: Section 8.5.2.

Problems 8.10 and 8.11 — AND and Filter

Listing 8.12: the empty list answers `true` (*vacuous truth*); **Listing 8.13**: filter the tail, then keep or drop the head.

Listing 8.12 The method `allPositive` of `AllPositive.java`.

```
    static boolean allPositive(List a) {
        if (a.empty()) {
            return true;                    // no element can break the rule
        }
        return a.head() >= 0 && allPositive(a.tail());
    }
```

Listing 8.13 The method `positives` of `Positives.java`.

```
    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;
    }
```

A second base case "a one-element list is fine" skips a check: `[4, -1]` becomes `true`. Stopping at the first negative element (`break` for `continue`) loses the 8 after the −5. Book: Sections 8.5.3 and 8.5.4.

Problem 8.12 — Is the List Sorted?

Listing 8.14: the combine step looks at **two** elements, so here two base cases are right.

Listing 8.14 The method `sorted` of `SortedCheck.java`.

```
    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());
    }
```

```
sorted([2, 3, -5, 8, -2]) = false
sorted([2, 3, 8])         = true
sorted([2, 5, 5, 8])      = true
sortedWrong([2, 5, 5, 8]) = false
```

"Increasing with duplicates allowed" means non-decreasing: `<=`. `sortedWrong` uses `<` and rejects `[2, 5, 5, 8]` — the most common single-character bug. Book: Section 8.5.5.

Problem 8.13 — Merging Two Sorted Lists

Listing 8.15: two lists, two base cases; each call moves the smaller head, so the work is linear, Equation (8.5).

Listing 8.15 The method `merge` of `MergeLists.java`.

```
    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()));
    }
```

$$\text{calls}(\text{merge}(a, b)) \leq |a| + |b|, \tag{8.5}$$

`merge([2,5,5,8], [9])` returns `b` whole and shared once `a` runs out. `mergeWrong` returns `nil()` there and loses the 9. `<=` keeps equal elements of `a` first: stability. Book: Section 8.5.6.

Problems 8.14 and 8.15 — Skip, Then Remove Duplicates

Listing 8.16 is a filter with the test "not equal to `x`"; **Listing 8.17** keeps the head and **reuses** `skip` to drop its later copies.

Listing 8.16 The method `skip` of `SkipElement.java`.

```
    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);
    }
```

Listing 8.17 The method `removeDuplicates` of `RemoveDuplicates.java`.

```
    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));
    }
```

`[3, 1, 3, 2, 1]` gives `[3, 1, 2]`: the input need not be sorted. The neighbor-only version is right for sorted input only. Book: Sections 8.5.7 and 8.5.8.

The Price of Simplicity

On n distinct elements the `skip` calls walk $n-1$, then $n-2$, … elements, Equation (8.6):

$$(n-1)+(n-2)+\cdots+1+0=\frac{n(n-1)}{2}, \tag{8.6}$$

For a million customer records that is about 5×10^{11} steps; a hash set (Chapter 9) is the right tool there.

Throwing an exception for the empty list is also wrong: an empty list has no duplicates, and its answer is the empty list. Exceptions are for situations with **no** sensible answer.

Check yourself: `removeDuplicates([4, 4, 1, 4, 1, 7])` *is* `[4, 1, 7]`*, and the* `skip` *calls walk 7 elements, well below 15. Book: Section 8.5.8.*

Problem 8.16 — An Exception-Safe Order Parser

> **PROBLEM.** Read `id;customer;service;kg` lines, keep every valid order, and report every bad line with its number and reason, *without stopping at the first bad line*. A missing file gives a clear message.

```
# 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
```

A bad line is expected: note it and carry on. A missing file fails the whole job: the caller decides. Book: Section 8.6.1.

Problem 8.16 — Who Reports, Who Handles

Table 8.4 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`

***Table 8.4**: the decisive choice is where each `try` goes — the one for `OrderFormatException` sits inside the loop, around one line. Book: Section 8.6.1.*

Problem 8.16 — Translate the Low-Level Error

Listing 8.18: `OrderFormatException` is **checked**, so every caller must think about it.

Listing 8.18 The end of `parseLine` in `LaundryOrderParser.java`: `NumberFormatException` becomes a message a person can act on.

```
        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);
```

`split(";", -1)` keeps empty trailing fields. Python translates `ValueError` with `raise … from None`, because our message already says everything. Book: Section 8.6.1.

Problem 8.16 — One Bad Line Costs One Line

Listing 8.19: the file in `try`-with-resources; the inner `try` records the message and keeps going.

Listing 8.19 The loop of `readOrders` in `LaundryOrderParser.java`.

```
        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
                }
            }
        }
```

`Files.newBufferedReader` throws `NoSuchFileException` for a missing file, and `main` turns it into `Cannot find the file ...`. Book: Section 8.6.1.

Problem 8.16 — The Report

```
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
```

The wrong reader puts one `try` around the whole loop: the first bad line ends the reading and `A107` is lost. And never write `catch (Exception e) {}`.

Three valid orders, 10.5 kg, five errors — one of each kind in the file; the comment and the blank line are skipped silently. Book: Section 8.6.1.

Problem 8.17 — An Array That Grows Itself

`GrowableIntArray` doubles its capacity when full; `get` must reject any index outside `0` to `size() - 1`. The class invariant is Equation (8.7):

$$0 \le size \le capacity = \texttt{data.length}, \quad \text{and only } \texttt{data}[0..size-1] \text{ holds values.} \tag{8.7}$$

Table 8.5 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)`	$0 \le i < size$	return `data[i]`	`IndexOutOfBoundsException`
`size()`, `capacity()`	none	report the two numbers	none

***Table 8.5**: the precondition of `get` is checked against `size`, not against the array length. Every public method carries Javadoc; `javadoc -quiet` prints nothing. Book: Section 8.6.2.*

Problem 8.17 — The Test at `index == size`

Listing 8.20: `add` copies into an array twice as long when full; `get` checks `index >= size`.

Listing 8.20 `add` and `get` of `GrowableIntArray.java`.

```
    public void add(int value) {
        if (size == data.length) {
            data = Arrays.copyOf(data, 2 * data.length);   // new, bigger array
        }
        data[size] = value;
        size++;
    }
    public int get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("index " + index + ", size " + size);
        }
        return data[index];
    }
```

```
add(7) -> [3, 4, 5, 6, 7]  size 5, capacity 8
getWrong(5) = 0   <- a value that was never added
```

`getWrong` tests `index > size`: `get(5)` reads a cell that exists (capacity 8) but was never added — a silent wrong answer. Doubling keeps the total copying below 2n. Book: Section 8.6.2.

Problem 8.18 — One Interface, Two Rules

> **PROBLEM.** Standard: Rp7,000 per kg. Express: Rp12,000 per kg plus Rp5,000. Every started kilogram is billed; zero or less is an error. One interface `PriceRule`, a method `quote(kg)`, tests for both.

By hand first: standard 2.3 kg, $\lceil 2.3 \rceil = 3$, 21 000; express, $3 \times 12\,000 + 5\,000 = 41\,000$; standard 0.1 kg, 7 000.

The rest of the program holds "some price rule" without knowing which; the rounding and the validation are written once. Book: Section 8.7.1.

Problem 8.18 — The Design

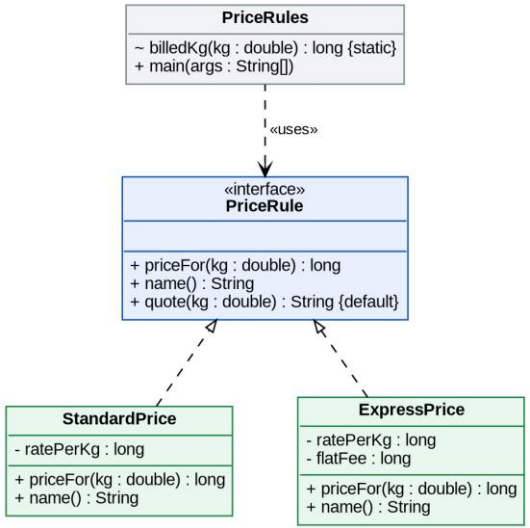


Figure 8.7 UML class diagram of Problem 8.18. The dashed lines with hollow triangles mean “implements”; `quote` is a default method of the interface.

Figure 8.7*: two abstract methods,* `priceFor` *and* `name`*, and one default method,* `quote`*, written in the interface in terms of the other two. Book: Section 8.7.1.*

Problem 8.18 — A Default Method

Listing 8.21: `quote` is written once for every present and future rule.

Listing 8.21 The interface `PriceRule` in `PriceRules.java`.

```
    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);
        }
    }
```

```
standard 2.3 kg = Rp21000
express 2.3 kg = Rp41000
wrong standard 2.3 kg = Rp16099
```

`(long) (7000 * 2.3)` gives 16099: two bugs, no rounding up and a truncated 16099.999 *…. Round up first, then multiply by an integer rate. Python runs one contract test class against both rules; `@Test` and `assertEquals` are the JUnit 6 form of `check`. Book: Sections 8.7.1 and 8.7.2.*

Problem 8.19 — Predict Before You Run

> **PROBLEM.** `Wash` (Rp7,000/kg), `Iron` (Rp5,000/kg) and `Express`, twice the wash rate, "(same day)". A final `price(kg)`, a `describe()`, and two overloads `label(Service)`, `label(Wash)`. Predict every line.

Overloading is resolved by the compiler, from the **declared** types; *overriding* by the running program, from the **object's class**.

Mixing them up is the most common misunderstanding of polymorphism. Book: Section 8.8.1.

Problem 8.19 — The Hierarchy

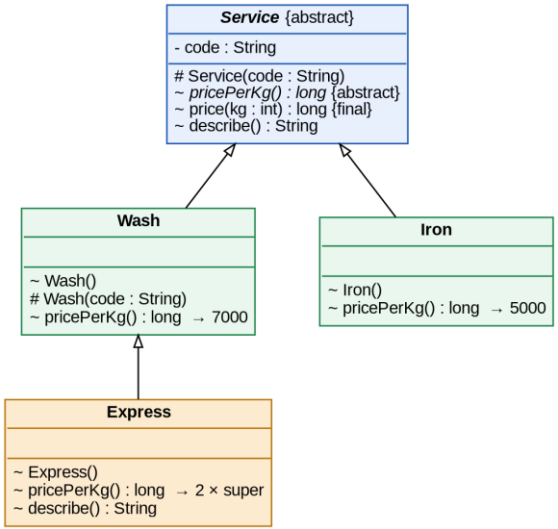


Figure 8.8 The service hierarchy of Problem 8.19. `Express` extends `Wash`, overrides `pricePerKg` and `describe`, and uses `super` to reach the `Wash` versions.

***Figure 8.8**: `price` is `final` in `Service` and calls the abstract `pricePerKg` — the template method pattern. `Express` extends `Wash` and reaches the `Wash` versions with `super`. Book: Section 8.8.1.*

Problem 8.19 — The Code to Trace

Listing 8.22: `describe` in `Service` calls `pricePerKg()`, which is dispatched on the object — even from superclass code.

Listing 8.22 From `ServiceDispatch.java`: the template in `Service`, and `Express` doubling the `Wash` rate.

```
        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";
        }
        @Override
        long pricePerKg() {
            return 2 * super.pricePerKg();      // twice the Wash rate
        }
```

Forgetting `super.` in `return 2 * super.pricePerKg();` makes the method call itself forever: `StackOverflowError`. Book: Section 8.8.1.

Problem 8.19 — The Hand Trace

Table 8.6 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)`

***Table 8.6**: line D is the lesson — `s` holds an `Express`, which is a `Wash`, yet `label(Service)` is chosen, because the compiler sees only `Service s`. Book: Section 8.8.1.*

Problem 8.19 — Run It and Compare

Listing 8.23 The `main` of `ServiceDispatch.java`: lines A to F.

```
        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
```

```
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)
```

Listing 8.23 *matches the trace line for line; a cast changes the declared type and therefore the overload (line F). Python's* `singledispatch` *chooses by run-time class and prints* `washing:` *for line D. Book: Section 8.8.1.*

Problem 8.20 — Two Copies Become One Hierarchy

`Player` and `Enemy` duplicate every field and `takeDamage`. By hand, three blows: villain 80 → 60, hero 100 → 90, villain 60 → 40 — and the original program **prints nothing**.

Table 8.7 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

***Table 8.7**: seven problems, each with its fix; the refactoring is extract superclass. Book: Section 8.8.2.*

Problem 8.20 — Before and After

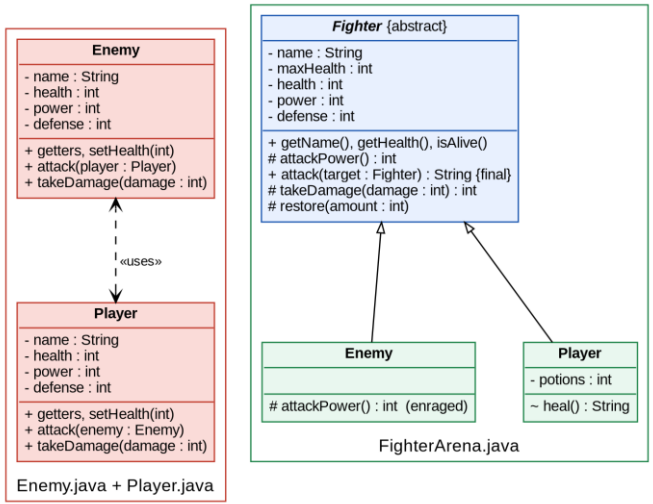


Figure 8.9 Before (left): two copies of the same fields and methods, each class using the other. After (right): one abstract `Fighter`, and subclasses that add only what differs.

***Figure 8.9**: one abstract `Fighter`; a `final` template method `attack`; a `protected` hook `attackPower()` where subclasses may change the rule. Book: Section 8.8.2.*

Problem 8.20 — A Final Template and a Hook

Listing 8.24: every attack follows the same steps; an enemy below half health is enraged.

Listing 8.24 From `FighterArena.java`: the `final` method `attack` and the hook overridden by `Enemy`.

```
    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)");
    }
    @Override
    protected int attackPower() {
        boolean enraged = getHealth() < getMaxHealth() / 2;
        return enraged ? getPower() + 10 : getPower();
    }
```

```
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)
```

At health 20 the villain is enraged: power 30, and the hero loses $30 - 10 = 20$*. No setter for health;* `Enemy extends Player` *would be a false is-a. Book: Section 8.8.2.*

Problem 8.21 — Four Ways In

`LeakyOrder` compiles and runs, yet outside code can break it in four ways. Ask of every field: "who can change this?"

Table 8.8 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*

***Table 8.8**: bugs 2 and 4 are both aliasing — two references to one mutable list, one held by a stranger. Book: Section 8.8.3.*

Aliasing, Drawn

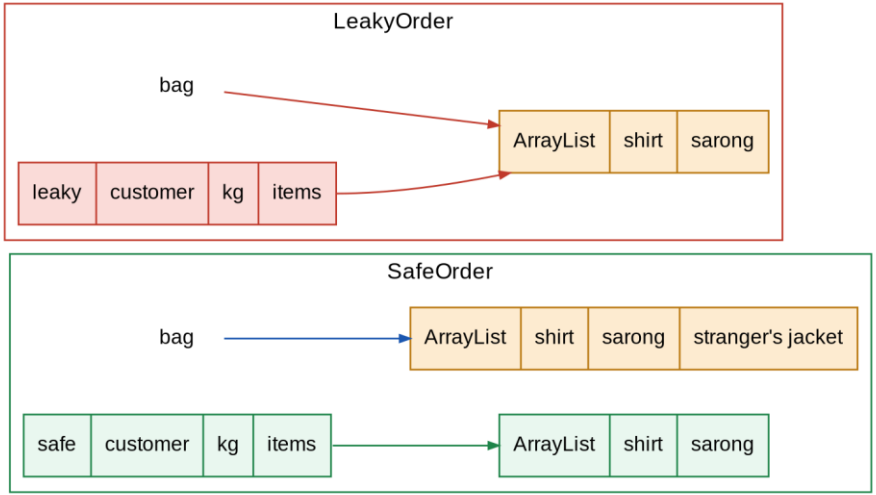


Figure 8.10 Aliasing. Top: `LeakyOrder` and the caller share one list, so either can change it. Bottom: `SafeOrder` copied the list, so the caller's later changes do not reach the order.

***Figure 8.10**: `SafeOrder` copied the list, so the caller's later changes do not reach it. Book: Section 8.8.3.*

Problem 8.21 — Copy In, Read-Only Out

Listing 8.25: validate in one place, copy the list in, hand out a read-only view.

Listing 8.25 From `OrderBugHunt.java`: the constructor and `getItems` of `SafeOrder`.

```
        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
        }
        java.util.List<String> getItems() {
            return Collections.unmodifiableList(items);      // read-only view OUT
        }
```

```
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]
```

`final` fixes the reference, not the list: `LeakyOrder.items` is already `final` and is emptied from outside. Python uses properties, a validating setter and tuples. Book: Section 8.8.3.

The Problems Compared: Trees and Lists

Table 8.9 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

***Table 8.9**, Problems 8.1–8.11: read down the last column — each wrong solution fails on one kind of input, rarely the example in the statement. Book: Section 8.9.*

The Problems Compared: the Rest

Table 8.10 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
8.12	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

***Table 8.10**, Problems 8.12–8.21: the strongest argument for Step 7 of the method — write the edge-case tests first. Book: Section 8.9.*

Profitina in Practice

One interface, many implementations: the scanner treats every signal through one interface, as the loop of Problem 8.18 treats every `PriceRule`; a new signal is a new class with the same contract tests.

Untrusted input without crashing: pages written by strangers and posted links are Problem 8.16 at scale — one bad item costs one item.

Encapsulation as a legal duty: consent, export and deletion under UU PDP 27/2022 need personal data changed in one controlled place, like `SafeOrder`; a release waits for over a thousand tests.

The product advises and its users decide: a score is a measurement with an explanation, not a guarantee. Book: Section 8.10.

Common Mistakes

Recursion: forgetting the missing subtree; ignoring a call's result; a second base case that skips a check; treating a signal value such as −1 as data.

Boundaries: `<` versus `<=`, "last used" versus "next free"; write the convention down and test exactly on it.

Exceptions and money: one `try` around everything; `catch (Exception e) {}`; money in floating point.

Classes: confusing overloading and overriding; forgetting `super`; inheritance for convenience; believing `final` makes an object immutable; testing only the statement's example.

Twelve mistakes are listed in the book, each with the problems that illustrate it. Book: Section 8.11.

Summary: Five Things to Remember

1. Solve a multi-class problem in eight steps: nouns and verbs, classes, owners, relationships, access, a drawing, tests with edge cases, one method at a time.

2. Write recursion with the delegation recipe; for lists, Equation (8.4) is the template of every solution.

3. Catch a bad record inside the loop, close files always, and translate low-level errors into messages people can act on.

4. Keep a class invariant with private fields, validation, defensive copies and read-only views.

5. Overloads are chosen at compile time, overrides at run time; extract a common superclass with a final template and a protected hook.

Every tempting wrong solution today passed the ordinary example and failed one edge case. Book: Section 8.12.

Next Time

Chapter 9 — Collections and Generics. Lists, sets and maps from the library, and type parameters that make them safe.

The hash set that Problem 8.15 was waiting for, and the `Set` that makes duplicate checks fast.

Before then: work Exercises 8.1, 8.5, 8.7 and 8.9, in Java and in Python.

The solutions to the exercises, with programs, are in the book. Book: Sections 8.13 and 8.14.

Chapter 1

Lecture 9 — Collections and Generics

Object-Oriented Programming (EF234302)

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*

Lecturer: Irfan Subakti — yifana@gmail.com

Book: Chapter 9, Collections and Generics. Every page of this handout ends with a pointer into the book, which is the primary reference; the handout is the chapter in miniature.

Today's Meeting

1. Two small tools: constants in interfaces, and recursion with a helper method (Section 9.2).

2. The Collections Framework and Python's collections: lists, sets, maps, queues, stacks and priority queues — how each stores its elements and what each operation costs (Sections 9.3–9.7).

3. Generics from zero: type parameters, generic methods, bounds, wildcards and type erasure — in Java, and in Python with `typing` and mypy (Section 9.8).

4. Order: `Comparable`, `Comparator` and Python's `key=` functions (Section 9.9).

5. Choosing a collection by counting its work, and the Profitina Laundry order book with four indexes over the same objects (Sections 9.10–9.13).

Book: Chapter 9, Sections 9.1 to 9.14.

Where We Left Off

Chapter 5 gave us arrays and `ArrayList<String>` — and promised that the `<String>` would be explained in Chapter 9. This is that chapter.

Chapter 6: an *interface* is a promise without a body; `List` is one, `ArrayList` implements it; `Comparable` promises a natural order.

Chapter 7: hierarchies, so one variable of type `Service` holds a wash, an iron or an express service. **Chapter 8:** recursive structures, an immutable integer list from `nil` and `cons`.

> **TODAY'S QUESTION.** A busy Saturday, a few hundred orders, four questions in one breath: *Where is order L-217? What has Bu Sari left with us? Which orders are ready? Which bag goes into the next free machine?* A loop over one big list answers all four — slowly. A **map**, a **set** per status and a **priority queue** answer each at once: the question chooses the collection.

Book: Section 9.1.

Constants in Interfaces

An interface may hold constants as well as method headers: `int LOSE = 0; int DRAW = 1; int WIN = 2;` inside `Measurable`.

Every field of an interface is **automatically** `public static final` — public (a public promise), static (an interface has no objects), final (no changing state).

A class that implements the interface inherits the names: inside `Team`, `r == WIN` needs no prefix. **Listing 9.1** shows both interfaces.

Listing 9.1 `InterfaceConstants.java` (excerpt): the interfaces `Measurable` and `Units` with their constants.

```
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
}
```

Two wins and a draw: $3 + 1 + 3 + 0 = 7$ points; eleven pounds are $11 \times 0.45359237 = 4.98951607$ kg, printed as `4.990`.

Book: Section 9.2.1 and Listing 9.1.

The Enum: a Safer Home

Interface constants are plain `int`s: nothing stops a caller from passing `7`. An **enum** declares a type with exactly the listed values, each an object with data and methods.

`Measurable.WIN = 5;` does not compile in Java (`cannot assign a value to static final variable WIN`). In Python it runs — and silently drops the team's score from 7 to 1, as **Listing 9.2** shows.

Listing 9.2 `interface_constants.py` (excerpt): a class attribute marked `Final` can be rebound at run time; an `Enum` member cannot.

```
    # 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)
```

```
after Measurable.WIN = 5: 1.0 points
AttributeError: cannot reassign member 'WIN'
```

> **mypy CHECKS ONLY ANNOTATED FUNCTIONS.** `Final` is a promise enforced by the type checker, not by Python. mypy reports both assignments — but only because `main` is declared `def main() -> None:`.

Book: Section 9.2.1 and Listing 9.2.

Recursion: Two Parts

A recursive method calls itself. Think *divide and conquer*: a smaller problem of the same kind, plus a little work.

The **base case**: the simplest version, answered directly; it stops the repetition.

The **recursive call**: the method calls itself with a *smaller* argument, moving towards the base case. Miss either and Java throws `StackOverflowError`, Python `RecursionError`.

The classic example, Equation (9.1), types into Java almost word for word:

$$F(0) = 0, \qquad F(1) = 1, \qquad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2 \tag{9.1}$$

Book: Section 9.2.2 and Equation (9.1).

Why the Standard Fibonacci Is Slow

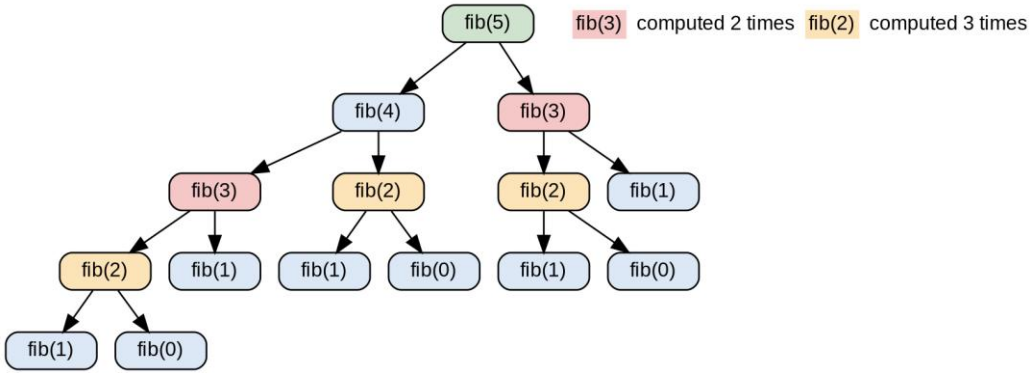


Figure 9.1 The calls made by the standard recursive `fibonacci(5)`: fifteen calls, because the same subproblems are solved again and again — `fib(3)` twice and `fib(2)` three times.

Figure 9.1: `fib(4)` *also* asks for `fib(3)`, and nobody remembers the first answer.

Book: Section 9.2.2 and Figure 9.1.

Counting the Calls

Let $C(n)$ be the number of calls made to compute $F(n)$: one for n itself, plus the calls of its two recursive calls.

$$C(0) = C(1) = 1, \qquad C(n) = 1 + C(n-1) + C(n-2) \quad \Rightarrow \quad C(n) = 2F(n+1) - 1 \tag{9.2}$$

Check by induction: $C(n) = 1 + (2F(n) - 1) + (2F(n-1) - 1) = 2F(n+1) - 1$. For $n = 5$: $2 \times 8 - 1 = 15$, the fifteen boxes.

For $n = 30$ Equation (9.2) gives $2 \times 1{,}346{,}269 - 1 = 2{,}692{,}537$ calls — for a number a child computes with thirty additions.

$F(n)$ grows like $\varphi^n/\sqrt{5}$ with $\varphi \approx 1.618$: every extra step multiplies the work by about 1.6.

Book: Section 9.2.2 and Equation (9.2).

A Helper Method Hides the Extra Parameters

Carry the two latest values along as parameters: one recursive call per step, about n calls — **tail recursion**.

But a caller who writes `fibonacci(10, 0, 0, 0)` gets a wrong answer. The **helper** `fib(n)` supplies the extra parameters (**Listing 9.3**).

Listing 9.3 `Fibonacci.java` (excerpt): the tail-recursive version and the helper method that callers use.

```
    // 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);
    }
```

`count >= n - 1`, not `==`: with `==` the call for $n = 1$ never reaches its base case. *A base case is only a base case if every chain of calls reaches it.*

Book: Section 9.2.3 and Listing 9.3.

The Call Counts, Measured

The output of the full program (book Listing 9.3) matches Equation (9.2) exactly: $2F(11) - 1 = 177$, $2F(21) - 1 = 21{,}891$, $2F(31) - 1 = 2{,}692{,}537$.

```
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
```

Python's `@lru_cache` makes the two-call definition *remember* every answer in a dictionary — linear time again (**Listing 9.4**).

Listing 9.4 `fibonacci.py` (excerpt): the standard definition made fast by `functools.lru_cache`.

```python
@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)
```

> **NO TAIL-CALL ELIMINATION.** Neither Java nor Python removes tail calls; Python stops at a depth of about 1000. For a million steps, write a loop.

Book: Section 9.2.3 and Listings 9.3 and 9.4.

The Collections Framework

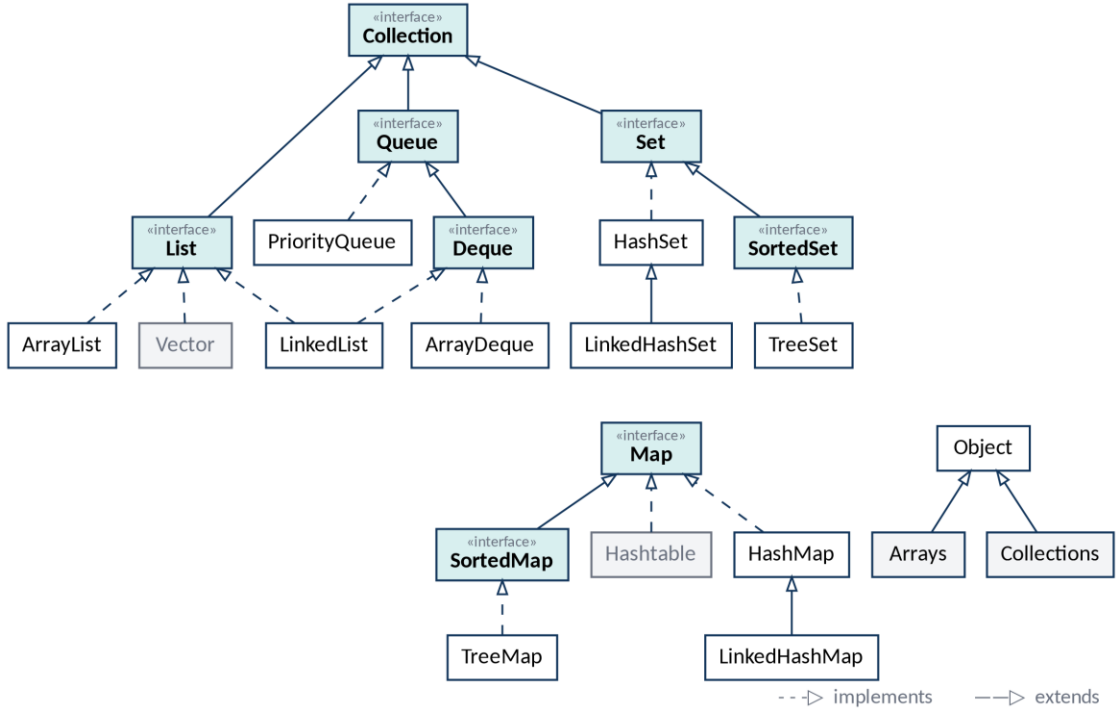


Figure 9.2 The core of Java's Collections Framework. Interfaces are marked «interface»; dashed arrows mean *implements* and solid arrows mean *extends*; the grey classes are historical. `Map` is not a `Collection`; `Arrays` and `Collections` hold static helper methods.

Figure 9.2: `Collection` → `List`, `Queue` (→ `Deque`), `Set`; `Map` stands apart; `Arrays` and `Collections` hold static helpers.

Book: Section 9.3 and Figure 9.2.

Three Families: Set, List, Map

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`

Table 9.1: a set answers *is it in me?*, a list keeps everything where you put it, a map answers *what goes with this key?*

Book: Section 9.3.1 and Table 9.1.

Program to the Interface

Write `List<String> myList = new ArrayList<>();`, not `ArrayList<String> myList = ...`. The rest of the code relies only on what every `List` can do.

If measurements later favor a `LinkedList`, change one word after `new`; nothing else can break.

Parameters too: a method that accepts `List<String>` accepts every list; one that demands `ArrayList<String>` needlessly rejects the others.

`new ArrayList<>()` — the **diamond** — lets the compiler fill in the type arguments (Java 7+).

Book: Section 9.3.2.

Python's Collections

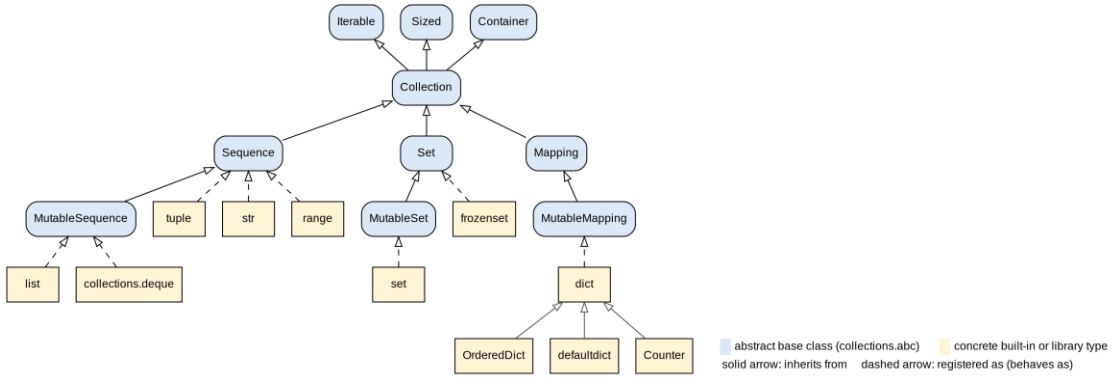


Figure 9.3 Python's collection types and the abstract base classes of `collections.abc`, the counterpart of Java's collection interfaces.

Figure 9.3: `list`, `tuple`, `set`, `dict` have their own syntax; `collections.abc` names the kinds, and built-ins are *registered* as behaving like them — duck typing made official.

Book: Section 9.3.3 and Figure 9.3.

Lists: the Everyday Operations

`add` returns `true` even for a duplicate; bulk operations take a whole collection (**Listing 9.5**).

Listing 9.5 `ListDemo.java` (excerpt): bulk operations and positions.

```
        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"));
```

`addAll` ≈ union, `removeAll` ≈ difference, `retainAll` ≈ intersection — but a list respects duplicates and order: `[and, us, us]`.

`add(0, "we")` inserts at the front, `set(1, "or")` replaces, `indexOf("us")` finds the *first* match: `[we, or, us, us]` `get(2) = us indexOf("us") = 2`.

Book: Section 9.4.1 and Listing 9.5.

Lists in Python

`append`, `insert`, indexing, `index`, `len` and `in` do the jobs of `add`, `add(i, x)`, `get`/`set`, `indexOf`, `size`, `contains`.

No `removeAll`/`retainAll`: the Pythonic way is a **list comprehension** (**Listing 9.6**).

Listing 9.6 `list_demo.py` (excerpt): comprehensions in place of `removeAll` and `retainAll`, and positions.

```
    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"))
```

`describe` serves a `list` and a `deque` with no declared interface — both have `pop` and `len`: duck typing.

Book: Section 9.4.1 and Listing 9.6.

ArrayList versus LinkedList

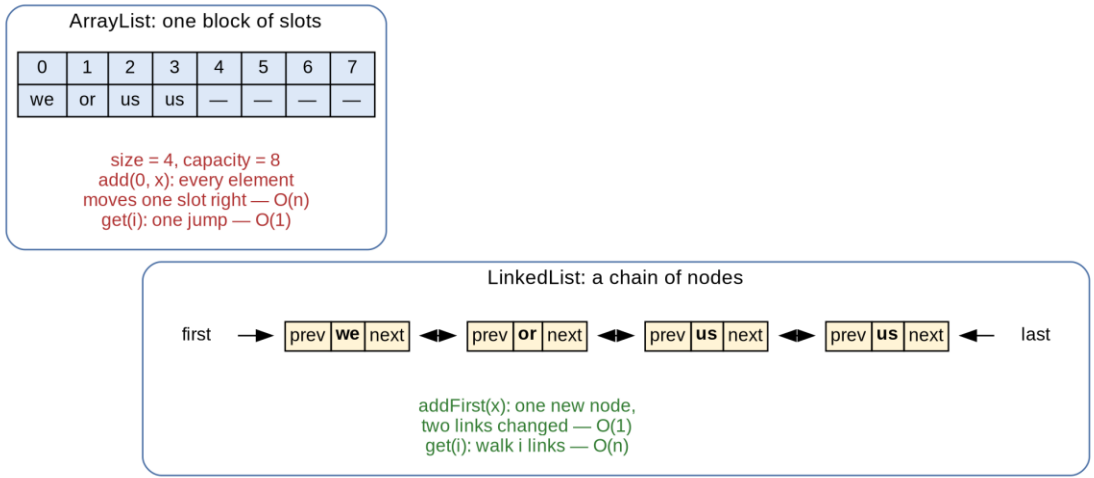


Figure 9.4 The same list stored two ways. An `ArrayList` keeps references in one block of consecutive slots, with spare capacity at the end; a `LinkedList` keeps a chain of nodes, each linked to its neighbors.

Figure 9.4: `get(i)` is $O(1)$ in the array and $O(n)$ in the chain; inserting at the front of an `ArrayList` shifts everything. n front inserts cost Equation (9.3):

$$0 + 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} \tag{9.3}$$

Book: Section 9.4.2, Figure 9.4 and Equation (9.3).

Which List Wins?

`ArrayList`: address of slot i in one step, so `get`/`set` are $O(1)$; appending is $O(1)$ on average. Front inserts: about 5×10^9 moves for $n = 100{,}000$.

`LinkedList`: inserting at either end is $O(1)$; but `get(i)` walks up to $n/2$ nodes, so an index loop over it costs $O(n^2)$.

In practice `ArrayList` almost always wins: neighboring slots are read fast; every linked node is a separate object scattered in memory.

For cheap work at *both* ends use `ArrayDeque`. Python's `list` is an `ArrayList`; its `deque` covers both ends.

Book: Section 9.4.2.

Removing While Walking: the Java Trap

Three ways to visit: by index, for-each (which uses an `Iterator`), and the `Iterator` by hand: `hasNext()`, `next()`.

Removing inside a for-each loop: the hidden iterator sees the list changed behind its back and throws — **fail-fast**. The cures are in **Listing 9.7**.

Listing 9.7 `IterationDemo.java` (excerpt): removing through the iterator itself, and with `removeIf`.

```
        // 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);

        // 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);
```

```
for-each + remove: ConcurrentModificationException
words now: [you, us, and, us]
iterator.remove: [you, and]
removeIf: [you, and]
```

Book: Section 9.4.3 and Listing 9.7.

The Python Trap Is Worse

In Python, removing inside a `for` loop raises **no error** — and leaves a wrong answer (**Listing 9.8**).

Listing 9.8 `iteration_demo.py` (excerpt): removing inside a `for` loop silently skips elements.

```
    # 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)
```

Trace `[you, us, us, and, us]`: at index 1 the first `us` is removed and the second moves into index 1; the loop moves to index 2 and never sees it. At index 3 `remove("us")` deletes the *first* `us` — the skipped one. The last `us` survives: `['you', 'and', 'us']`.

> **KEY INSIGHT.** Never change the *structure* of a collection while a loop walks it, except through the walker itself. Loop over a copy (`words[:]`), use a comprehension, `Iterator.remove` or `removeIf`. Replacing an element in place is harmless.

Book: Section 9.4.3 and Listing 9.8.

Sets: Six Arrivals, Three Orders

`add(3)` on a set that already holds 3 returns `false`. Set algebra: $\{1,2,3\} \cup \{1,\ldots,7\}$, $\{1,\ldots,7\} \setminus \{4,\ldots,10\} = \{1,2,3\}$, $\{1,2,3\} \cap \{2,3,4,5\} = \{2,3\}$.

The same six arrivals in three kinds of set, **Listing 9.9**:

Listing 9.9 `SetDemo.java` (excerpt): the same names in a `HashSet`, a `LinkedHashSet` and a `TreeSet`, and nearest-element queries.

```
        // 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"));
```

```
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]
```

Book: Section 9.5.1 and Listing 9.9.

Sets in Python

Operators: `|` union, `-` difference, `&` intersection, `^` symmetric difference, each with an in-place form such as `|=`.

A `set` of strings iterates in an order that changes *from run to run* (hash salting). When order matters, ask for it: `sorted(s)` ≈ `TreeSet`; `list(dict.fromkeys(items))` ≈ `LinkedHashSet` (**Listing 9.10**).

Listing 9.10 `set_demo.py` (excerpt): first-seen order, sorted order and a ceiling found with `bisect_left`.

```
    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')]}")
```

A `frozenset` is immutable and hashable: it can be a key of a `dict` or an element of another set.

Book: Section 9.5.1 and Listing 9.10.

Hashing: Compute, Don’t Search

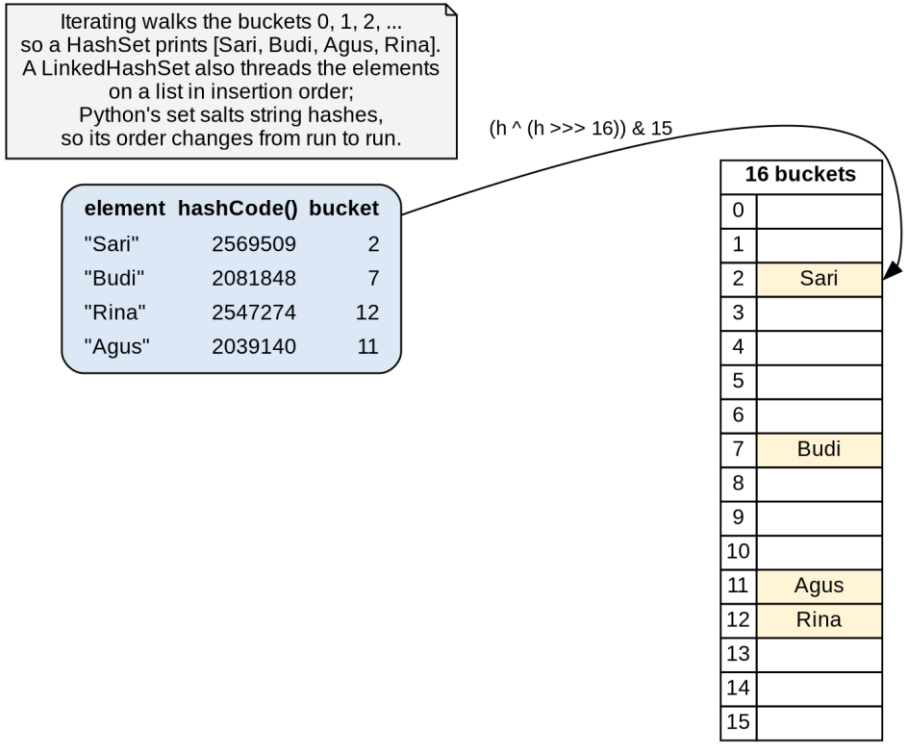


Figure 9.5 A `HashSet` of four names. Each name's hash code selects one of 16 buckets, so `contains` looks in one bucket only; iteration walks the buckets in index order, which explains the printed order `[Sari, Budi, Agus, Rina]`.

Figure 9.5: `contains("Rina")` computes bucket 12 and compares only there; iteration walks buckets 2, 7, 11, 12.

Book: Section 9.5.2 and Figure 9.5.

Worked Example: Where Does “Sari” Go?

A `String`’s hash code: $s_0 \cdot 31^{k-1} + s_1 \cdot 31^{k-2} + \cdots + s_{k-1}$, with 32-bit wrap-around.

`"Sari"`: $83 \cdot 31^3 + 97 \cdot 31^2 + 114 \cdot 31 + 105 = 2{,}472{,}653 + 93{,}217 + 3{,}534 + 105 = 2{,}569{,}509$.

A new set has $m = 16$ buckets; bucket $=$ `(h ^ (h >>> 16)) & 15` $= 2$ for `"Sari"`.

The average chain length is the **load factor** Equation (9.4); Java doubles the table when $n > 0.75\,m$, which gives Equation (9.5):

$$\alpha = \frac{n}{m} \tag{9.4}$$

$$T_{\text{hash}}(n) = O(1 + \alpha) = O(1) \tag{9.5}$$

Book: Section 9.5.2 and Equations (9.4) and (9.5).

A TreeSet Keeps Order

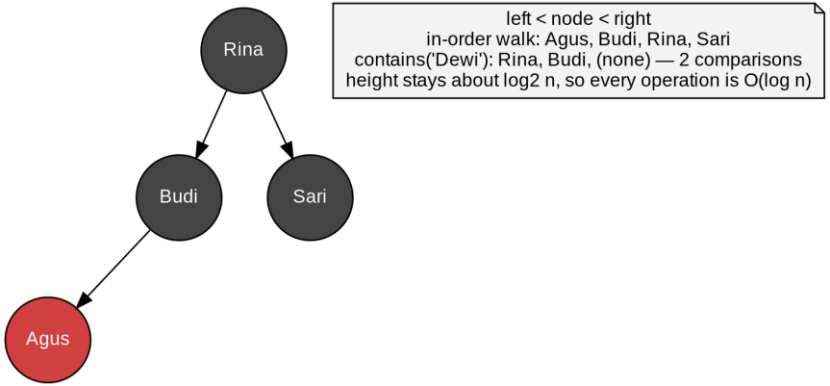


Figure 9.6 The `TreeSet` of the same four names: a red-black search tree. An in-order walk visits the names in sorted order; a search follows one path from the root.

Figure 9.6: to find `"Dewi"`, go left at `Rina`, right at `Budi` — nothing there, absent after two comparisons. A red-black tree keeps Equation (9.6):

$$h \leq 2\log_2(n+1) \tag{9.6}$$

Book: Section 9.5.3, Figure 9.6 and Equation (9.6).

Maps: Keys to Values

`put` **returns the previous value** (or `null`); `get` returns `null` for an absent key; `getOrDefault` gives a default.

`containsKey` is a hash lookup; `containsValue` looks at every value, $O(n)$.

Counting in one line: `counts.merge(t, 1, Integer::sum)` (**Listing 9.11**).

Listing 9.11 `MapDemo.java` (excerpt): counting with `merge` in a `LinkedHashMap`, then the same counts in a `TreeMap`.

```
        // 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));
```

```
LinkedHashMap {WASH=3, IRON=2, WASH_IRON=1}
TreeMap       {IRON=2, WASH=3, WASH_IRON=1}
```

Book: Section 9.6.1 and Listing 9.11.

Python's dict

`d[key]` on a missing key raises **`KeyError`**; use `d.get(key, default)`.

A `dict` keeps **insertion order** (Python 3.7+): a `LinkedHashMap` by default.

Three ways to count, and `defaultdict(list)` for grouping — the counterpart of `computeIfAbsent` (**Listing 9.12**).

Listing 9.12 `map_demo.py` (excerpt): counting with `get`, with `Counter`, and grouping with `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))
```

```
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)]
```

Book: Section 9.6.1 and Listing 9.12.

Queues, Stacks, Priority Queues

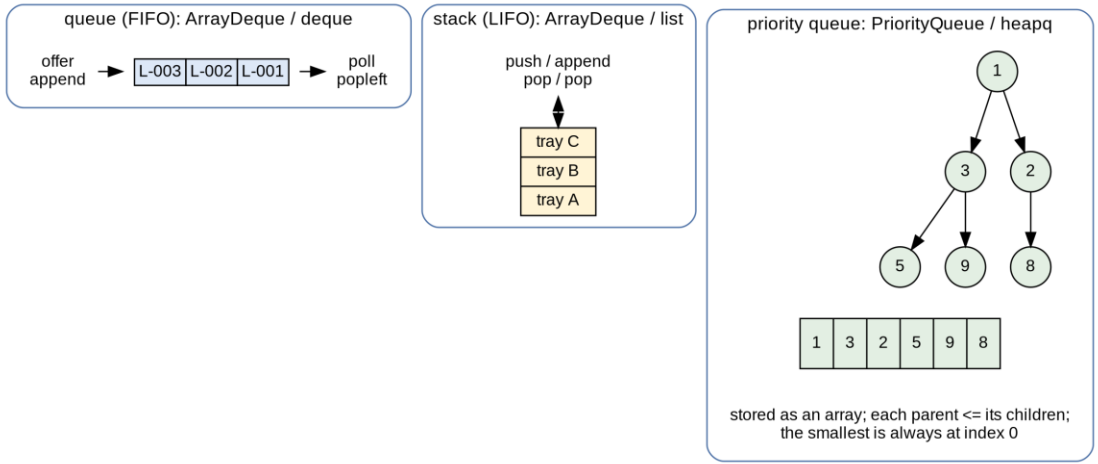


Figure 9.7 Three disciplines for taking elements out. A queue releases the oldest element (FIFO), a stack the newest (LIFO), and a priority queue the smallest, which a binary heap keeps at index 0 of its array.

Figure 9.7: FIFO releases the oldest, LIFO the newest, a priority queue the smallest — kept at index 0 of a heap array.

Book: Section 9.7 and Figure 9.7.

Worked Example: a Binary Heap

`ArrayDeque` is a circular array: queue *and* stack, both ends $O(1)$. The priority queue is a **binary heap** (**Listing 9.13**).

Listing 9.13 `QueueDemo.java` (excerpt): a `PriorityQueue` of six numbers, printed and then polled until empty.

```
        // 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());
```

```
heap array [1, 3, 2, 5, 9, 8]
poll order 1 2 3 5 8 9
```

`[1, 3, 2, 5, 9, 8]` is *not* sorted: children of index i sit at $2i+1$ and $2i+2$ and every parent is no larger than its children — so index 0 is the minimum. `offer`/`poll` $O(\log n)$, `peek` $O(1)$.

Book: Section 9.7 and Listing 9.15.

Queues in Python

`collections.deque`: `append` and `popleft`, both $O(1)$. A plain `list` is the usual stack.

`heapq` applies the heap rules to an ordinary list — the same array `[1, 3, 2, 5, 9, 8]` (**Listing 9.14**).

Listing 9.14 `queue_demo.py` (excerpt): `heapq` as a priority queue, and negated numbers for largest-first.

```
    # 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))))

    # 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))
```

THE MOST COMMON QUEUE BUG IN PYTHON. `list.pop(0)` shifts every remaining element: $O(n)$ per call. Use `deque.popleft()`.

Book: Section 9.7 and Listing 9.16.

Generics: the Definition

> **DEFINITION.** A **generic class** is written with one or more **type parameters**, placeholder names for types: `class GList<T>`. A use supplies a **type argument**, `GList<Integer>`, and the compiler checks every use as if `T` had been replaced by `Integer`. A **generic method** declares its own type parameters before its return type: `static <T> T lastOf(List<T> list)`.

One class that does not care what kind of object it holds — as long as they are all of one kind — and a compiler that holds every user to the kind they chose.

Conventions: `T` type, `E` element, `K` and `V` key and value.

Book: Section 9.8.

Life Before Generics

Before Java 5 (2004) a `List` held `Object`s. The raw list accepts `42` among the names; the error appears only at the cast, far from the mistake (**Listing 9.15**).

Listing 9.15 `RawTypes.java` (excerpt): a raw `List` accepts an `Integer`, and the cast fails at run time.

```
    @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");
            }
        }
    }
```

With `List<String>`, `names.add(42)` is refused by `javac`: *incompatible types: int cannot be converted to String*. Errors move from run time, where users meet them, to compile time, where programmers do.

Book: Section 9.8.1 and Listing 9.17.

A Generic Class: GList

Chapter 8's list of *integers*, written once for any element type (**Listing 9.16**).

Listing 9.16 `GListDemo.java` (excerpt): the generic class `GList<T>`, its two constructors and the generic method `nil()`.

```
// 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;
    }

    public static <T> GList<T> nil() {                  // a generic method
        return new GList<>();
    }
```

`toString` hands the work to `toStringHelper(list, accumulator)` — the recursion-with-helper pattern of Section 9.2.3.

Book: Section 9.8.2 and Listing 9.19.

One Class, Two Type Arguments

`GList<Integer>` gives `[]`, `[10]`, `[50, 10]`; the *same class* holds `String`s, and `towels.head()` returns a `String` without a cast.

```
myGList: []
myGList2: [10]
myGList3: [50, 10]
towels: [white, blue], head white, tail [blue], size 2
IllegalStateException: Trying to access tail of an empty list
```

`nil()` is static, so it cannot use the class's `T`: it declares its own, `public static <T> GList<T> nil()`, and the compiler infers `T` from the context.

> **RAW RETURN TYPES SWITCH OFF CHECKING.** `public static GList nil()` compiles, with warnings, and quietly disables checking for every caller. Compile with `javac -Xlint:all` and treat every "unchecked" or "rawtypes" warning as a bug.

Book: Section 9.8.2 and Listing 9.19.

The Same Class in Python

`T = TypeVar("T")` creates a type variable; `class GList(Generic[T])` declares the parameter; `GList[int]`, `GList[str]` are types mypy understands (**Listing 9.17**).

Listing 9.17 `glist_demo.py` (excerpt): `TypeVar`, `Generic[T]`, and the two ways to build a list, `nil()` and `cons()`.

```
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
```

Python 3.12 adds `class GList[T]:`; the listing uses the older form so that it runs on Python 3.10 and 3.11.

Book: Section 9.8.2 and Listing 9.20.

Generic Methods, and a Pair at Last

`static <A, B> Pair<B, A> swap(Pair<A, B> p)`: "for any types `A` and `B` …". The compiler **infers** `A` and `B` from the argument (**Listing 9.18**).

Listing 9.18 `GenericMethods.java` (excerpt): the generic methods `swap` and `lastOf`, and `fibonacciPair` returning a `Pair<Long, Long>`.

```
    // <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());
    }
```

`fibonacciPair(10)` now prints `(55, 34)`, with parts named `getFirst()` and `getSecond()` instead of `[0]` and `[1]`.

Book: Section 9.8.3 and Listing 9.21.

Bounded Type Parameters

`a.compareTo(b)` inside `GList<T>` does not compile: `T` could be *any* class. Say "any `T` that can be compared with other `T`s": `<T extends Comparable<T>>` (**Listing 9.19**).

Listing 9.19 `GList2Demo.java` (excerpt): the bounded generic method `max`.

```
    // 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;
    }
```

`max(List.of(3, 9, 4))` is 9, `max(List.of("wash", "dry", "iron"))` is `wash`; `GList2<Object>` is refused: *type argument Object is not within bounds of type-variable T*.

Python: `TypeVar("T", bound=SupportsLessThan)`, a **protocol** with `__lt__` (book Listing 9.24); mypy rejects a list of plain `object`s where it is built.

Book: Section 9.8.4 and Listings 9.23 and 9.24.

Is a List a List?

It seems obvious. It is false. If `List<Animal> animals = cats` compiled, `animals.add(new Dog(...))` would put a dog into the list of cats.

So generic types are **invariant**, Equation (9.7) ($<:$ reads "is a subtype of"):

$$Cat <: Animal \quad \not\Rightarrow \quad G\langle Cat\rangle <: G\langle Animal\rangle \tag{9.7}$$

Arrays are **covariant** — a `Cat[]` *is* an `Animal[]` — so `pets[0] = dog1` compiles and fails at run time with `ArrayStoreException: Dog`.

Book: Section 9.8.5 and Equation (9.7).

The Two Escape Routes: Wildcards

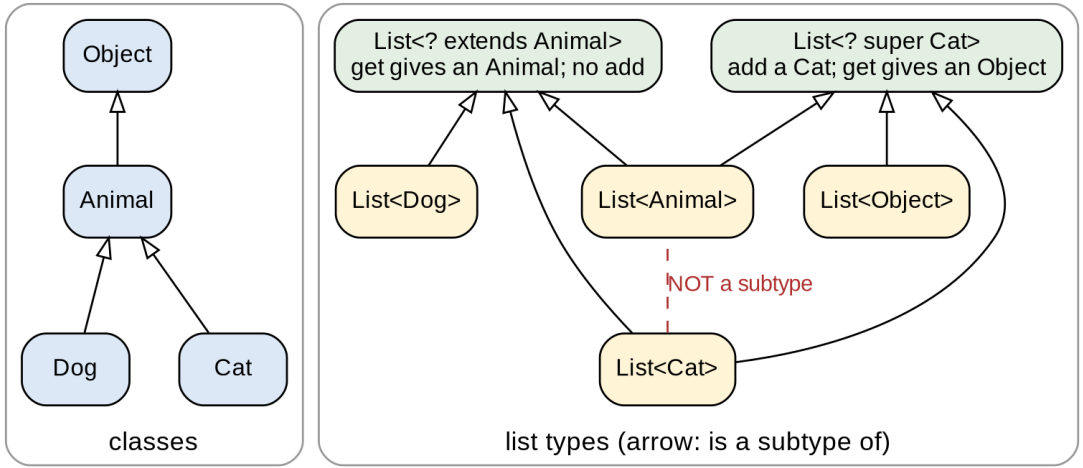


Figure 9.8 Subtyping among classes (left) and among list types (right). `List<Cat>` is not a subtype of `List<Animal>`, but both are subtypes of `List<? extends Animal>`, and `List<Cat>`, `List<Animal>` and `List<Object>` are all subtypes of `List<? super Cat>`.

Figure 9.8: `? extends Animal` accepts lists of cats and dogs for **reading**; `? super Cat` accepts lists of cats, animals and objects for **writing**.

Book: Section 9.8.5 and Figure 9.8.

PECS: Producer extends, Consumer super

Table 9.2 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

Table 9.2: a parameter the method only *takes from* is a producer, `? extends T`; one it only *puts into* is a consumer, `? super T`.

Book: Section 9.8.5 and Table 9.3.

Wildcards at Work

`names` only reads, so it takes `List<? extends Animal>`; `addAll` reads `src` and writes `dest` (**Listing 9.20**).

Listing 9.20 `WildcardDemo.java` (excerpt): a consumer parameter `? super T` and a producer parameter `? extends Animal`.

```
    // 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();
    }
```

With `T = Cat` or `T = Dog`, `dest` may be a `List<Animal>` or a `List<Object>`. `Collections.addAll` and `Collections.copy` are declared the same way.

Python: `list[T]` is invariant, `Sequence[T]` is covariant — `Sequence[Animal]` plays `List<? extends Animal>`. Ignore the rule and a dog ends up among the cats; mypy says *"list" is invariant — Consider using "Sequence"* (book Listing 9.26).

Book: Section 9.8.5 and Listings 9.25 and 9.26.

Type Erasure

Java checks type arguments, inserts the casts, then **erases** them: at run time `List<String>` is just `ArrayList` (**Listing 9.21**).

Listing 9.21 `ErasureDemo.java` (excerpt): one class at run time, and heap pollution through a raw parameter.

```
    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)");
        }
```

```
same class at run time: true
class name: java.util.ArrayList
size 2, contents [Siti, 42]
ClassCastException at get(1)
```

No `new T()`, no `new T[n]`, no `instanceof List<String>`, no `List<int>`. `Collections.checkedList` refuses the wrong element at the door. Python goes further: type arguments were never part of the running program (book Listing 9.28).

Book: Section 9.8.6 and Listings 9.27 and 9.28.

Generics in Java and Python

Table 9.3 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

Table 9.3: `javac` enforces generics always; mypy enforces Python's hints when you run it.

Book: Section 9.8.7 and Table 9.4.

Comparable or Comparator?

`Comparable<T>`: the class defines its *natural ordering* with `compareTo`. `Comparator<T>`: a separate object defines *an* ordering from outside — by age, by name, largest first.

Contract: negative, zero or positive; and consistent (if a before b and b before c, then a before c).

Listing 9.22 `MyCollection.java` (excerpt): `AgeComparator`, a comparator written as a class.

```
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
        }
    }
}
```

Listing 9.22 for `s1` (age 20) and `s2` (age 12): $20 - 12 = 8$, positive, so `s1` comes after `s2`. Safe for ages — not in general.

Book: Section 9.9.1 and Listing 9.29.

Comparators for Real Use

Since Java 8 comparators are *built* from the field you sort by, and combined (**Listing 9.23**).

Listing 9.23 `SortingDemo.java` (excerpt): natural order, visits descending, and city then name descending.

```
        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()));
```

```
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)
```

The three customers with four visits stay in name order: Java's object sort is **stable**.

Book: Section 9.9.2 and Listing 9.31.

Two Comparator Traps

A `TreeSet` with a comparator uses *the comparator* to decide equality: three customers with four visits count as one — `[Dewi(4), Agus(7), Budi(9)]`. Add tie-breakers until "compares equal" means "the same element".

Subtraction overflows. For $a = -2{,}000{,}000{,}000$ and $b = 2{,}000{,}000{,}000$, $a - b = -4 \times 10^9$ wraps to $-4{,}000{,}000{,}000 + 2^{32} = 294{,}967{,}296$ — *positive*.

```
TreeSet by visits [Dewi(4), Agus(7), Budi(9)]
subtraction says low vs high = 294967296, Integer.compare says -1
```

Use `Integer.compare(a, b)` or `Comparator.comparingInt`, always.

Book: Section 9.9.2 and Listing 9.31.

Python Sorts by Key

Python's sort takes a **key function** of *one* argument; `cmp_to_key` adapts a Java-style comparison (book Listing 9.30).

Mixed directions on *strings*: no number to negate — use stability. Sort by the *minor* key first, then the *major* key (**Listing 9.24**).

Listing 9.24 `sorting_demo.py` (excerpt): visits descending with `attrgetter`, then two stable sorts for city ascending, name descending.

```
    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}")
```

Python's integers never overflow: `low - high` is simply −4,000,000,000.

Book: Sections 9.9.1–9.9.2 and Listings 9.30 and 9.32.

What Each Operation Costs in Java

Table 9.4 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 9.4: $O(1)$ bounded, $O(\log n)$ one more step each time n doubles, $O(n)$ proportional to n. Python's `list`, `deque`, `set`, `dict` and `heapq` cost the same as their Java twins (book Table 9.6).

Book: Section 9.10 and Tables 9.5 and 9.6.

One contains, Three Collections

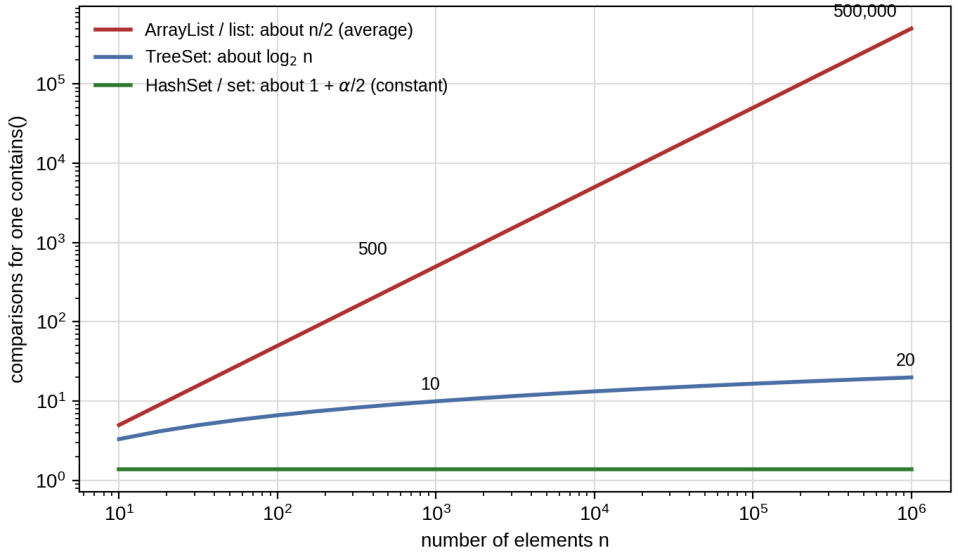


Figure 9.9 Comparisons made by one `contains` as n grows (both axes logarithmic): about $n/2$ for a list, about $\log_2 n$ for a balanced tree, and a small constant for a hash table with load factor $\alpha \leq 0.75$.

Figure 9.9: at a million elements a list needs half a million comparisons, a tree 20, a hash table one or two.

Book: Section 9.10 and Figure 9.9.

Choosing with Four Questions

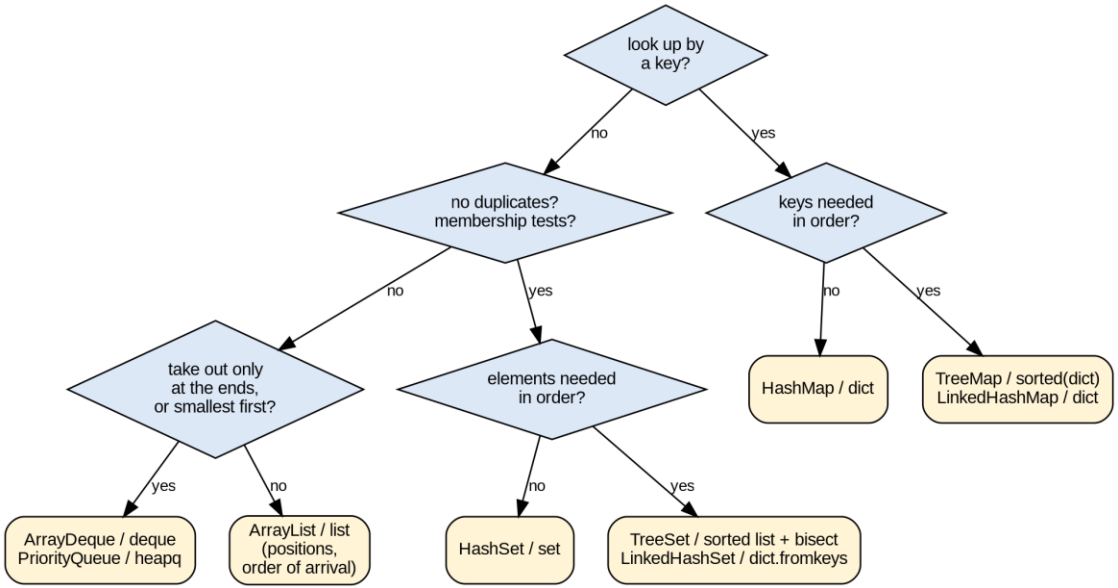


Figure 9.10 Choosing a collection with four questions. Each leaf names the Java class and the Python type that answer them.

Figure 9.10: look up by key? no duplicates? only the ends, or smallest first? If every answer is “no”, use a list. Measure before optimizing.

Book: Section 9.10 and Figure 9.10.

Profitina Laundry: Four Indexes

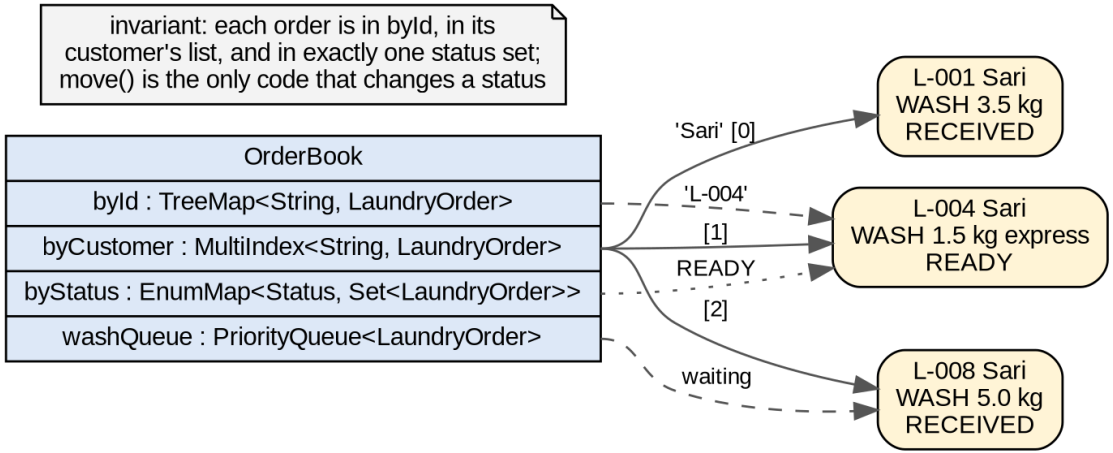


Figure 9.11 The order book: four structures, one set of `LaundryOrder` objects. The class keeps an invariant: every order appears in `byId`, in its customer's list, and in exactly one status set.

Figure 9.11: a `TreeMap` by id, a `MultiIndex` by customer, an `EnumMap` of `LinkedHashSet`s by status, a `PriorityQueue` for washing — all over the *same* objects.

Book: Section 9.11 and Figure 9.11.

The Invariant That Keeps Them Honest

Several indexes must never disagree: an order that is `READY` but still in the `WASHING` set gives a wrong answer, silently.

The class keeps Equation (9.8), true after every public method:

$$\sum_{s \in Status} |byStatus[s]| = |byId| \quad \text{and} \quad o \in byStatus[o.status] \text{ for every order } o \tag{9.8}$$

Only one private method, `move`, ever changes a status — and it updates the index in the same breath (**Listing 9.25**).

Listing 9.25 `LaundryOrderBook.java` (excerpt): `move`, the one place where the status index changes.

```
    // 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);
    }
```

Removal happens *before* the field changes, while the order can still be found in its old set.

Book: Section 9.11, Equation (9.8) and Listing 9.33.

Express First, Then First Come

The queue's comparator compares `!o.isExpress()` (`false` sorts before `true`), then the arrival number (**Listing 9.26**).

Listing 9.26 `LaundryOrderBook.java` (excerpt): the four structures of `OrderBook`, all `private final`, and a constructor that gives every status its own empty set.

```
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<>());
        }
    }
```

`receive` refuses a duplicate id *before* touching any structure, so a refused order leaves no trace.

Book: Section 9.11 and Listing 9.33.

The Same Queue in Python

`heapq` compares with `<`, and a `LaundryOrder` has none. So each entry is a tuple `(not order.express, order.arrival, order)`; the unique arrival numbers mean the comparison never reaches the order (**Listing 9.27**).

Listing 9.27 `laundry_order_book.py` (excerpt): `receive` pushes a priority tuple; `start_next_wash` pops it.

```
    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))

    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
```

Ordered sets: a `dict` with `None` values; `MultiIndex(Generic[K, V])` returns a *tuple*, an immutable snapshot (book Listing 9.34).

Book: Section 9.11 and Listing 9.34.

The Run

```
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
```

Express `L-002, L-004, L-007` first; iron-only `L-003` never enters the queue. The counts add up to $4 + 1 + 2 + 1 = 8$, as Equation (9.8) demands.

Book: Section 9.11 and Listing 9.33.

Worked Example: the Revenue Report by Hand

Prices follow Equation (9.9): WASH Rp7,000, IRON Rp5,000, WASH_IRON Rp10,000 per kg; express adds half.

$$p = \mathrm{round}(w \cdot r) + e \cdot \left\lfloor \frac{\mathrm{round}(w \cdot r)}{2} \right\rfloor, \qquad e = \begin{cases} 1 & \text{express} \\ 0 & \text{otherwise} \end{cases} \tag{9.9}$$

Budi: round$(5.0 \times 10{,}000) = 50{,}000$, express 75,000; plus $4.0 \times 7{,}000 = 28{,}000$; total 103,000.

Sari: $3.5 \times 7{,}000 = 24{,}500$; express 1.5 kg: $10{,}500 + 5{,}250 = 15{,}750$; $5.0 \times 7{,}000 = 35{,}000$; total 75,250.

Agus 42,000, **Dewi** $25{,}000 + 12{,}500 = 37{,}500$, **Rina** $2.0 \times 5{,}000 = 10{,}000$. Total $267{,}750 = 10{,}000 + 145{,}250 + 112{,}500$ by service.

Book: Section 9.11 and Equation (9.9).

Profitina in Practice

Profitina scans a brand's website and measures how ready it is to be quoted by AI answer engines (GEO). It is written in Python.

A **set** of canonical URLs answers "fetched it already?" in expected $O(1)$; pages to fetch wait in a **queue**, or a **priority queue** when some matter more.

Maps from page to measurements, from signal to score, from week to score (read sorted, exported as CSV), and "N new" counters per channel.

A hint such as `dict[str, list[Signal]]` states what a function returns; a suite of well over a thousand tests is the release gate. The measurements advise; the user decides.

Book: Section 9.12.

Common Mistakes

Declaring the class, not the interface; removing inside a for-each loop; `get(i)` over a `LinkedList` or `pop(0)` on a Python `list`; `contains` on a list for frequent membership tests.

Relying on hash order; changing a field that `hashCode` or `compareTo` uses while the object sits in a set, map or tree.

Subtraction comparators; a comparator that calls different objects equal in a `TreeSet`; raw types and ignored "unchecked" warnings.

Expecting `List<Cat>` to be a `List<Animal>`; believing Python hints are checked at run time; `d[key]` or `map.get(key).method()` on a missing key; indexes that drift apart.

Book: Section 9.13.

Summary: Five Things to Remember

1. A collection answers a question you ask often; program to the interface, `List`, `Set`, `Map`, `Queue`, `Deque`.

2. Hash sets and maps: expected $O(1)$ because the load factor stays below 0.75; trees: sorted, $O(\log n)$; lists: positions, $O(n)$ search; heaps: smallest first in $O(\log n)$.

3. Never change a collection's structure while walking it — except through the iterator.

4. Generics move errors to compile time; they are invariant, wildcards restore flexibility (PECS), and Java erases them while Python never had them at run time.

5. Order comes from `Comparable`, `Comparator` or a `key=`; sorts are stable; one class and one invariant keep several indexes honest.

Book: Section 9.14.

Next Time

Chapter 10 — Graphical User Interfaces and Event Handling. The program stops being in charge: a window waits, and the user decides what happens next.

Swing and Tkinter side by side; the event loop explained from zero; listeners, layouts, painting, inner classes and lambdas — the lambdas we met today in `removeIf` and comparators.

Before the meeting: Exercises 9.1, 9.4, 9.7 and 9.8, and read Sections 10.1 to 10.4.

Book: Section 9.15 and Chapter 10.

CHAPTER 1

Lecture 10 — Graphical User Interfaces and Event Handling

Object-Oriented Programming (EF234302)

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*

Lecturer: Irfan Subakti — yifana@gmail.com

Book: Chapter 10, Graphical User Interfaces and Event Handling. Every page of this handout ends with a pointer into the book, which is the primary reference; the handout is the chapter in miniature.

Today's Meeting

1. Who is in charge: from the console to the window; AWT, Swing and Tkinter; the first window and its self-test (Sections 10.2–10.5).

2. The event loop explained from zero, and a toy loop you can run (Section 10.6).

3. Listening to events, and laying out a window with layout managers, `pack` and `grid` (Sections 10.7–10.8).

4. Painting, and the listener styles: inner classes, anonymous classes, lambdas, closures — and what each remembers (Sections 10.9–10.10).

5. The thread rule, and the Profitina Laundry order-entry window in Swing and Tkinter (Sections 10.11–10.14).

Book: Chapter 10, Sections 10.1 to 10.15.

Where We Left Off

Every program so far started at the top of `main`, did its work in the order written, printed, and stopped. Even with input (Chapter 5), the program was in charge.

Chapter 7 gave Profitina Laundry polymorphic service classes; Chapter 9 stored many objects in type-safe collections.

The staff at the counter do not use a terminal. They use a **window**, and they click wherever they like, in whatever order they like.

> **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 hands control to a loop that waits for things to happen.

Book: Section 10.1.

From the Console to the Window

The console asks five questions in a fixed order; change your mind about *express* after the weight, and there is no way back.

The window shows all five boxes at once; the program is a set of small handlers: *when the weight changes, recompute the price; when* Add order *is clicked, validate…*

> **DEFINITION.** An **event** is anything the program may want to respond to: a click, a key press, a mouse movement, a window uncovered. An **event handler** (a **listener** in Java, a **callback** in Python) is a method the toolkit calls when that event occurs. A program organized this way is **event-driven**; the library calls your code — **inversion of control**.

Consequences: **handlers must be short**; **state lives in objects**, not in the position in the code; the window is **described**, not narrated.

Book: Section 10.2.

The Layers Under a Window

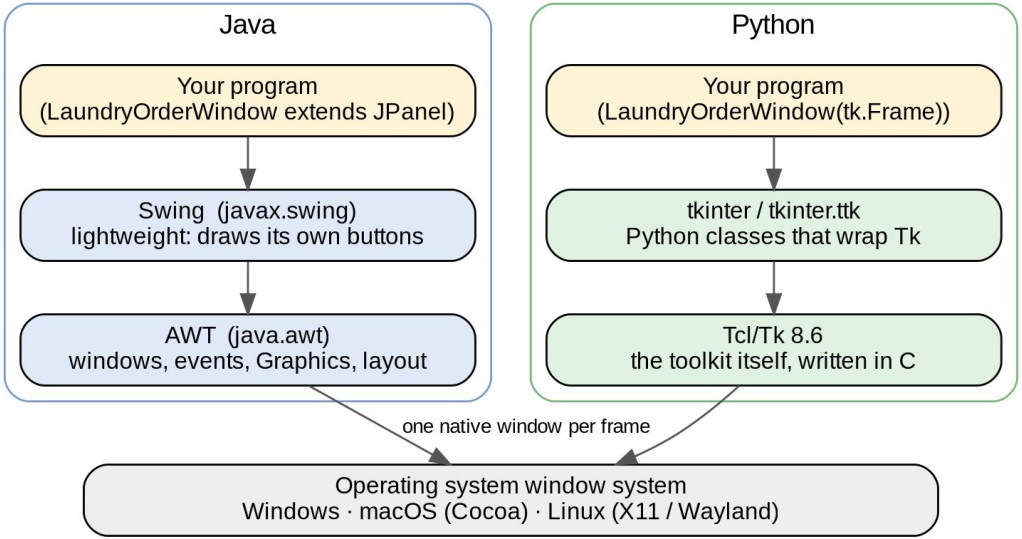


Figure 10.1 The layers under a window. In Java, your panel is built from Swing components, which rest on AWT, which asks the operating system for a native window. In Python, `tkinter` is a thin layer of Python classes over the Tcl/Tk toolkit.

Figure 10.1: Swing rests on AWT, which asks the operating system for a native window; `tkinter` is a thin Python layer over Tcl/Tk.

Book: Section 10.3 and Figure 10.1.

AWT, Swing and Tkinter

AWT (`java.awt`, JDK 1.0): native, *heavyweight* peers. Still the foundation: windows, event classes and listener interfaces (`java.awt.event`), `Graphics`, colors, layout managers.

Swing (`javax.swing`, JDK 1.2): *lightweight* components that paint themselves; class names start with **J** — `JFrame`, `JButton`, `JLabel`. *In a Swing program, use only the J classes.*

Tkinter: Python's interface to Tk; one root `Tk` per program, every widget created *inside* a parent given as the first argument; `tkinter.ttk` adds themed widgets.

> **NO TKINTER?** On Debian/Ubuntu `sudo apt install python3-tk`; on Fedora `sudo dnf install python3-tkinter`. `ModuleNotFoundError: No module named 'tkinter'` means this, not your code.

Book: Sections 10.3.1–10.3.3.

Two Class Hierarchies

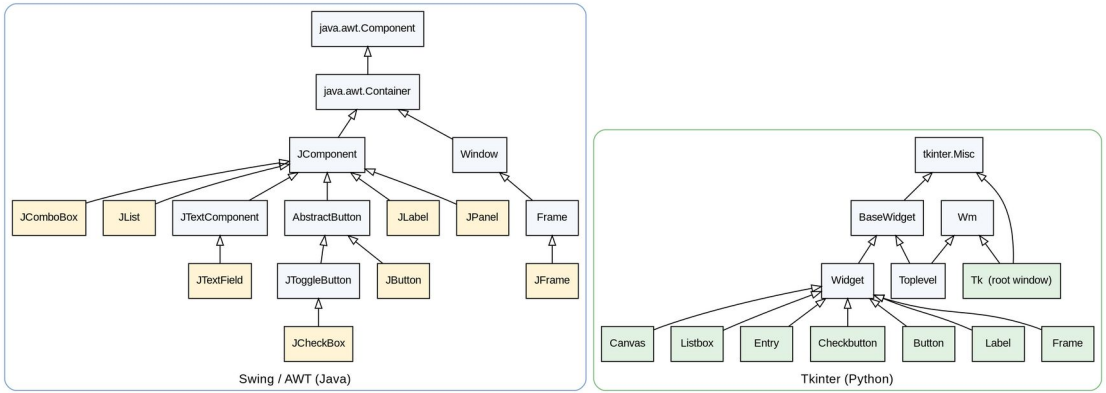


Figure 10.2 Part of the class hierarchies of Swing/AWT (left) and Tkinter (right). Shaded classes are the ones this chapter uses. Arrows point from a subclass to its superclass.

Figure 10.2: `JFrame` descends from AWT's `Window`; every other Swing component from `JComponent`; every component is also a `Container`. Tkinter's design is flatter.

Book: Section 10.3.2 and Figure 10.2.

Your First Window in Swing

Listing 10.1 holds the six steps that almost every Swing program performs.

Listing 10.1 `FirstWindow.java` (excerpt): the six steps of `createAndShow`.

```
    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);
    }
```

`EXIT_ON_CLOSE`: without it, closing only *hides* the frame and the program seems to hang. `pack()` beats `setSize`: preferred sizes depend on fonts.

`main` passes `createAndShow` to `SwingUtilities.invokeLater` and ends — yet the program keeps running: the **event dispatch thread** lives on.

Book: Section 10.4 and Listing 10.1.

The Self-Test: a Window Without a Screen

Run with `--selftest`, every program builds its components exactly as usual, drives them from code, prints what it finds, and never opens a window.

`System.setProperty("java.awt.headless", "true")` comes first: lightweight components work, a `JFrame` would throw `HeadlessException`.

```
headless: true
components: 1
label text: Hello, Profitina Laundry!
on the EDT: true
self-test passed
```

Tk has no headless mode: on a Linux server, run `xvfb-run -a python3 first_window.py --selftest`.

> **KEY INSIGHT.** A self-test mode costs a dozen lines and pays for itself many times over: it forces the window to be built by a method, checks behavior from code, and lets a server (or a book) run your GUI.

Book: Section 10.4.1.

The Same Window in Tkinter

`tk.Tk()` is the root; `tk.Label(root, ...)` is created *inside* it; `pack()` places it and sizes the window (Listing 10.2).

Listing 10.2 `first_window.py` (excerpt): building the window inside the root.

```
def build(root):
    root.title("First window")
    label = tk.Label(root, text="Hello, Profitina Laundry!", padx=20, pady=10)
    label.pack()
    return label
```

Listing 10.3 `first_window.py` (excerpt): `main`, which hands control to the event loop.

```
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
```

Listing 10.3: `root.mainloop()` hands control to Tk's event loop; it returns only when the window is closed. The self-test instead calls `root.withdraw()` and never shows the window.

Book: Section 10.4 and Listing 10.2.

Components, Containers, Panels

A window is a **tree**: the frame at the root, containers as inner nodes, simple controls as leaves.

A `JFrame` holds several panes; your components go into the **content pane** — or build your own `JPanel` and install it with `frame.setContentPane(pane)`.

Good practice, used for every window of the chapter: *each significant part of a window is a class that extends `JPanel`* (Tkinter: `tk.Frame`), with its components in fields and built in the constructor.

Reading a window as a tree is the key to laying it out: every container places its own children and nothing else.

Book: Sections 10.5.1–10.5.3.

The Component Tree of a Real Window

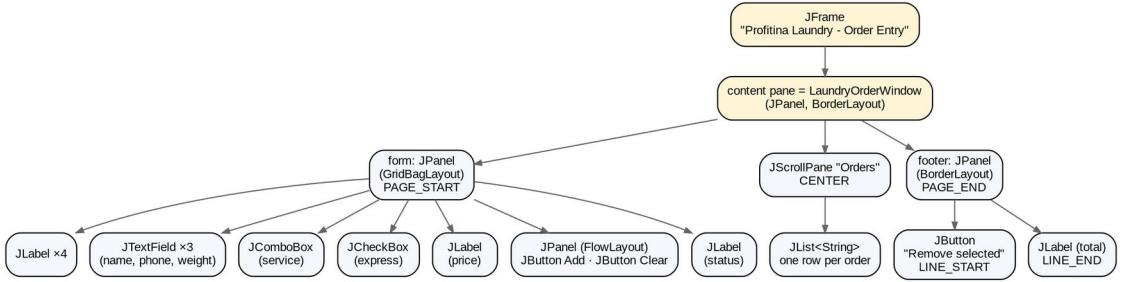


Figure 10.3 The component tree of the Profitina Laundry order-entry window. Each container arranges only its own children, with its own layout manager.

Figure 10.3: the laundry order-entry window of Section 10.12 — a form, a scrolling list and a footer, each with its own layout.

Book: Section 10.5.3 and Figure 10.3.

The Event Loop

The operating system turns a press, a key or an uncovered window into an **event** and queues it for the program that owns the window.

Inside every toolkit runs 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
```

Line 3: waiting, not even using the processor. Lines 4–7: **dispatch**. Line 7 is the *only* place your code runs after `main` — and while it runs, nothing else happens.

Book: Sections 10.6.1–10.6.2 and Pseudocode 10.1.

The Life of an Event

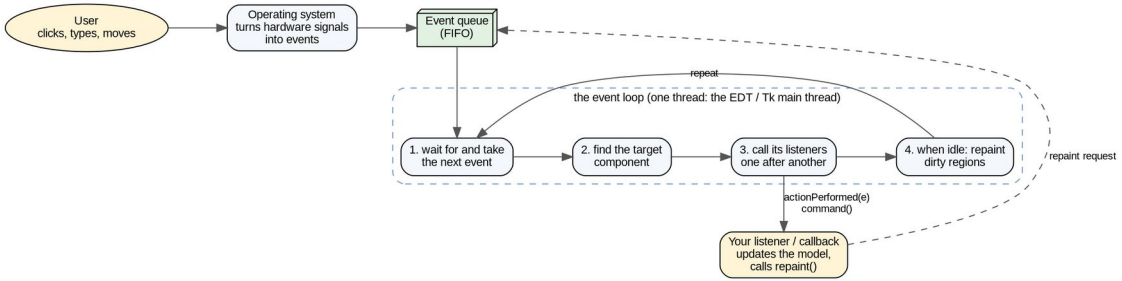


Figure 10.4 The life of an event. The operating system turns the user's actions into events and queues them; the event loop takes them one at a time, calls the listeners registered for them, and repaints when idle. A listener that changes what should be shown only *requests* a repaint.

Figure 10.4: a listener that changes what should be shown only *requests* a repaint; the loop repaints when idle (lines 8–9).

Book: Section 10.6.2 and Figure 10.4.

An Event Loop You Can Read

Pretend widgets with listener lists, a queue, and the loop of Pseudocode 10.1 almost line for line (Listing 10.4).

Listing 10.4 `toy_event_loop.py` (excerpt): the hand-written event loop — wait, take, dispatch.

```
    # 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}")
```

Registration runs nothing: `button.add_listener(on_click)` only stores the function, to be called later.

Listeners share state through objects: `typed` is a one-element list (Java: a one-element array) — Section 10.10 explains why.

Book: Section 10.6.3 and Listing 10.4.

Worked Example: Tracing the Toy Loop

```
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
```

$4 \times 7000 = 28{,}000$: the click listener prints nothing; it *enqueues a repaint*. The next pass finds the repaint already queued and dispatches it before asking the user for anything. Then $3 \times 7000 = 21{,}000$, total $49{,}000$; `close` breaks the loop.

Book: Section 10.6.3 and Listings 10.3 and 10.4.

Listening to Events

In Java, three steps: **declare** (`implements ActionListener`), **register** (`addActionListener`), **define** (`actionPerformed(ActionEvent e)`).

The button does not care what class you are — only that you promised `actionPerformed`. Chapter 6's interfaces at work.

The event object says what just happened: `e.getSource()` (which component), `e.getPoint()`, `e.getX()` for the mouse, `e.getKeyCode()` for keys.

Book: Sections 10.7 and 10.7.2.

A Button and Its Listeners

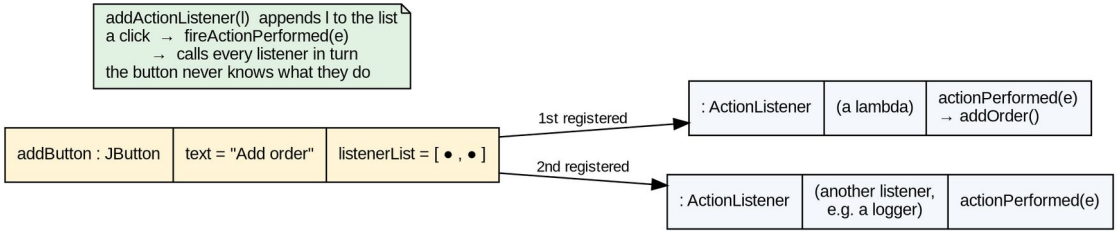


Figure 10.5 A button and its listeners. The button keeps a list; registration appends to it; a click calls each listener's `actionPerformed` in turn. The button never knows what its listeners do.

Figure 10.5: a subject that notifies a list of observers without knowing what they do — the **Observer** pattern, the backbone of every toolkit.

Book: Section 10.7 and Figure 10.5.

Reading a Text Field

The numbered comments of Listing 10.5 mark the changes compared with a panel that only displays; change (1) is the import of `ActionEvent` and `ActionListener` from `java.awt.event`.

Listing 10.5 `EchoPanel.java` (excerpt): declare, register, define.

```
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());
    }
```

Book: Section 10.7.1 and Listing 10.5.

The Echo Window

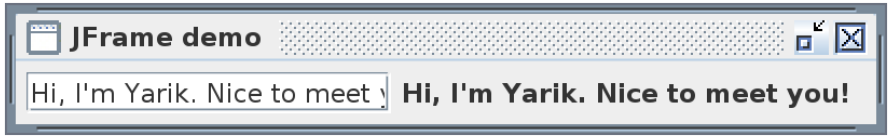


Figure 10.6 The echo window after the user typed a greeting into the text field and pressed Enter: the listener has copied the text into the label.

Figure 10.6: after Enter. The self-test calls `t.postActionEvent()` — exactly what Enter does — and finds one listener on the field.

Book: Section 10.7.1 and Figure 10.6.

Tkinter: bind and the Event Object

No listener interfaces: `widget.bind("<Return>", callback)` attaches any callable to an **event sequence** — `"<Button-1>"`, `"<B1-Motion>"`, `"<KeyPress-a>"` (Listing 10.6).

Listing 10.6 `echo_panel.py` (excerpt): a `tk.Frame` subclass that binds Enter to a bound method.

```
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())
```

Tk calls the callback with an **event object**: `event.widget`, `event.x`, `event.y`, `event.keysym`.

A withdrawn window receives no key events, so the self-test calls `panel.on_enter(None)` itself; buttons offer `invoke()`.

Book: Section 10.7.1 and Listing 10.6.

Item Events: Check Boxes

`implements ItemListener`, `cb.addItemListener(this)`, `itemStateChanged(ItemEvent e)` — one listener for both boxes (Listing 10.7).

Listing 10.7 `FruitChoice.java` (excerpt): one item listener that asks which box changed.

```java
    @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");
        }
    }
```

```
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
```

`doClick()` performs a click through exactly the same path as a mouse, listeners included. The event says what just happened; `getSource()` says which component.

Book: Sections 10.7.2–10.7.3 and Listing 10.7.

The Check-Box Window



Figure 10.7 The check-box window after *Mango* was ticked: the item listener rewrote the explanation label.

Figure 10.7: after *Mango* was ticked. In Tkinter each `Checkbutton` stores its state in a `BooleanVar` and gets its own `command` lambda — the lambda *is* the answer to "which box?" (book Listing 10.8).

Book: Section 10.7.3 and Figure 10.7.

The Four Kinds of Events

Table 10.1 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 (check box 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)`

Table 10.1: a panel may listen to several kinds at once — `ShadeBox` handles action and mouse events.

Book: Section 10.7.4 and Table 10.2.

Layout by Rules, Not Pixels

"At pixel (192, 82), 104 pixels wide" is wrong on the next computer: other fonts, other languages, a wider window.

Java: every container has a **layout manager**, which uses each child's *preferred*, *minimum* and *maximum* sizes. Tkinter: a **geometry manager** — `pack`, `grid`, `place`.

FlowLayout, the default of a `JPanel`: left to right like words, new row when full, preferred sizes kept (Figure 10.8).



Figure 10.8 Five buttons in a `FlowLayout`: each keeps its preferred size, and the row flows from left to right.

Book: Sections 10.8 and 10.8.1, Figure 10.8.

BorderLayout: Five Regions



Figure 10.9 The five regions of a `BorderLayout`. The center takes all the space the four edges leave over.

Figure 10.9: the default of a content pane. Top and bottom take the full width, left and right the remaining height, the center everything left over.

Book: Section 10.8.2 and Figure 10.9.

BoxLayout: One Row or One Column

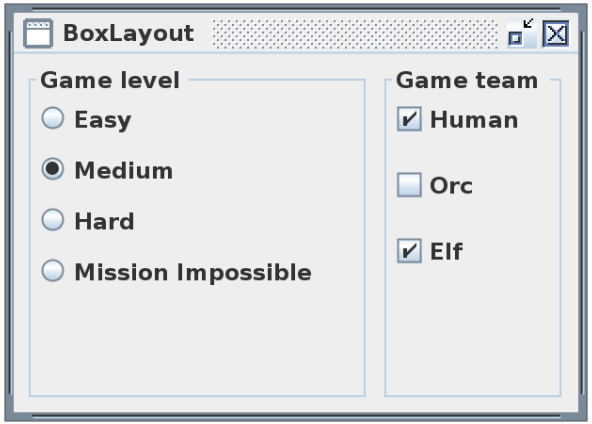


Figure 10.10 Two vertical boxes inside a horizontal box. The radio buttons on the left belong to one `ButtonGroup`, so only one can be chosen; the check boxes on the right are independent.

Figure 10.10: `Box.createVerticalBox()`, struts for fixed gaps, glue for springs.

Book: Section 10.8.3 and Figure 10.10.

GridBagLayout: the Most Flexible

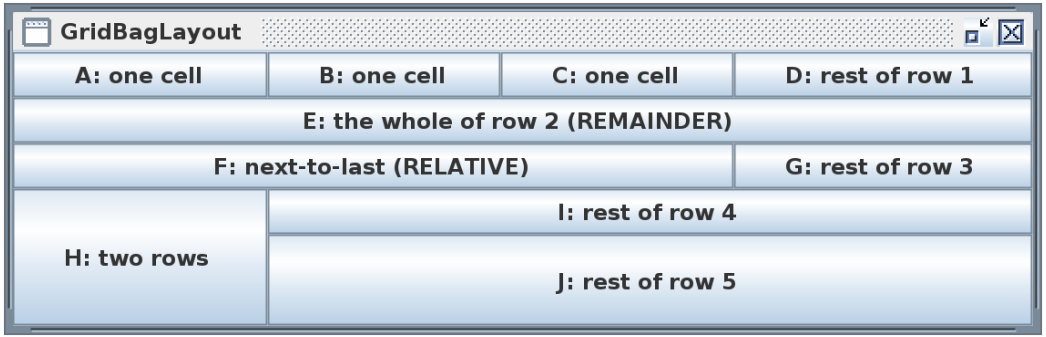


Figure 10.11 A `GridBagLayout` in which buttons span one cell, the rest of a row, or two rows, as set by `gridwidth`, `gridheight` and the weights.

Figure 10.11: a grid of cells with `GridBagConstraints` — `gridx`, `gridwidth`, `weightx`, `fill`, `anchor`. Forgetting to reset a reused constraint is the classic bug; hide it in a helper.

Book: Section 10.8.4 and Figure 10.11.

Tkinter: pack, grid and place

`pack(side=..., fill=..., expand=...)` stacks widgets against the sides of their parent, in the order packed.

`grid(row=..., column=..., sticky="ew")` plus `columnconfigure(1, weight=1)`: `GridBagLayout` without the constraints object.

`place(x=..., y=...)`: absolute coordinates — rarely a good idea.

> **NEVER MIX pack AND grid IN ONE PARENT.** Tk 8.6 refuses: `TclError: cannot use geometry manager grid inside . which already has slaves managed by pack`. One manager per parent; nest frames when you need both.

Book: Section 10.8.5.

Worked Example: Watching BorderLayout Work

The self-test of `LayoutTour.java` sizes each panel, calls `doLayout()`, and prints the bounds (book Listing 10.9; an excerpt of its output):

```
  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
```

Top and bottom: full width 460 at height 25. Left and right keep their preferred widths 187 and 120 and stretch to the remaining height $150 - 25 - 25 = 100$. The center gets $460 - 187 - 120 = 153$.

In Tkinter, `pack(side=LEFT)` does **not** wrap: at 230 pixels, *Long-Named Button 4* and *5* get no space at all.

Book: Section 10.8.6 and Listing 10.9.

Choosing a Layout

Table 10.2 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)`

Table 10.2: real windows are built by **nesting** — a `BorderLayout` for the shape, a `grid` form in one region, a row of buttons inside the form.

Book: Section 10.8.6 and Table 10.3.

Painting: the Coordinate System

Override `protected void paintComponent(Graphics g)`: `g` is a pen already pointed at the panel. The origin is the **top-left**, and y grows **downward** (Figure 10.12).

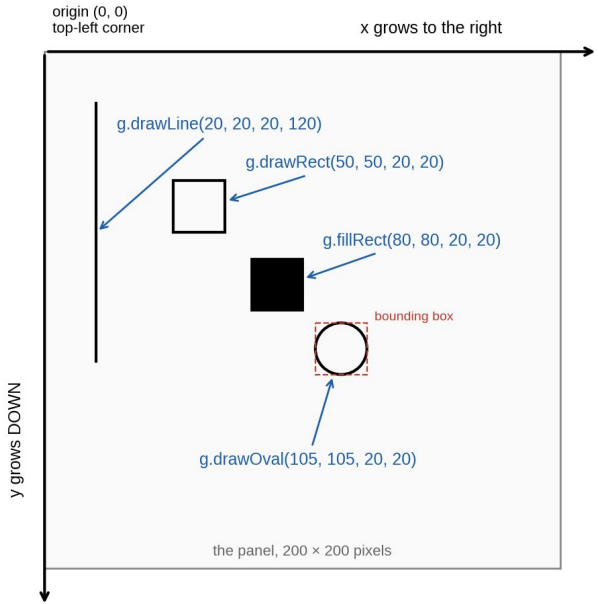


Figure 10.12 Swing's coordinate system: the origin is the top-left corner and y grows downward. An oval is specified by its bounding box, not by its center.

Book: Section 10.9.1 and Figure 10.12.

Color, and Who Calls paintComponent

RGB: red, green, blue, each 0–255; equal amounts give gray. Equation (10.1) counts the colors:

$$256 \times 256 \times 256 = 256^3 = 16,777,216 \tag{10.1}$$

You never call `paintComponent`. Swing calls it when the window appears, is restored, uncovered or resized; it must paint everything from the panel's fields.

After changing the state, call `repaint()`: it only marks the panel dirty; the loop paints later (line 9 of Pseudocode 10.1). First line of every `paintComponent`: `super.paintComponent(g)`.

Book: Sections 10.9.2–10.9.3 and Equation (10.1).

A Box That Darkens and Lightens

`ShadeBox` listens to action *and* mouse events; two buttons and clicks on the box itself (Listing 10.8).

Listing 10.8 `ShadeBox.java` (excerpt): painting from the state, and `shade` with `clamp`.

```
    @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));
    }

    private void shade(int delta) {
        level = clamp(level + delta);
        darken.setEnabled(level > 0);
        lighten.setEnabled(level < 255);
        repaint();  // ask for paintComponent to run soon
    }
```

`actionPerformed` asks `e.getSource() == darken` — reference equality: *the very same object*. `shade` changes the state and calls `repaint()`; it never draws.

Book: Section 10.9.4 and Listing 10.11.

Worked Example: How Many Clicks to Black?

`clamp` computes Equation (10.2), from the inside out:

$$v' = \min\left(255, \max\left(0, v + \delta\right)\right) \tag{10.2}$$

Each click subtracts 10 and the last one is clamped, so white to black takes Equation (10.3):

$$\left\lceil \frac{255}{10} \right\rceil = \lceil 25.5 \rceil = 26 \tag{10.3}$$

After 25 clicks the level is $255 - 250 = 5$; the 26th gives -5, clamped to 0, and *Darken* is disabled. The painted pixel is `#0a0a0a`: $0\text{a}_{16} = 10$ after one click in the box.

```
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
```

Book: Section 10.9.4 and Equations (10.2) and (10.3).

The Shading Window



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.

Figure 10.13: the box is black and *Darken* is disabled. `MouseListener` forces five methods on the class, four of them empty — Section 10.10 removes that nuisance.

Book: Section 10.9.4 and Figure 10.13.

A Label That Explains



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.

Figure 10.14: one `setText` at the limits saves the user from wondering why a button stopped working.

Book: Section 10.9.4 and Figure 10.14.

The Tkinter Canvas: Retained Mode

`create_rectangle(...)` creates an **item** Tk remembers and redraws; change it with `itemconfig` — no `paintComponent`, no `repaint()` (Listing 10.9).

Listing 10.9 `shade_box.py` (excerpt): a canvas item, a click bound to the item itself, and `shade` changing the item.

```
        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)
```

> **KEY INSIGHT.** Swing (immediate mode): *state in fields, pixels in* `paintComponent`, `repaint()` *to connect them*. Tk canvas (retained mode): *create items once, then change the items*.

Book: Section 10.9.5 and Listing 10.12.

Inner Classes, Anonymous Classes, Lambdas

DEFINITION. An **inner class** is declared inside another class without `static`. Every inner object is created inside an outer object and keeps a hidden reference to it, so it can use the outer object's fields and methods — even `private` ones; the reference is written `Outer.this`.

An **anonymous class** declares and instantiates in one expression: `new ActionListener() { ... }`. Extending an **adapter** (`MouseAdapter`) overrides only the methods you need.

A single-method interface is a **functional interface**; a **lambda** implements it: `darken.addActionListener(e -> shade(-10));`. A **method reference**: `this::resetCount`.

Book: Sections 10.10.1–10.10.2.

Five Styles Side by Side

Listing 10.10: the panel itself, a named inner class, an anonymous class, a lambda and a method reference — plus lambdas made in a loop.

Listing 10.10 `ListenerStyles.java` (excerpt): five ways to write an action listener, and lambdas created in a loop.

```
        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);
        }
```

Counts: 1, 3, 6, 10, then +10, +20, +30 give 20, 40, 70; *Reset* returns to 0. The inner class compiles to `ListenerStyles$AddListener.class`, the anonymous one to `ListenerStyles$1.class`; the lambda gets a hidden class at run time.

Book: Section 10.10.3 and Listing 10.13.

What Listeners Remember

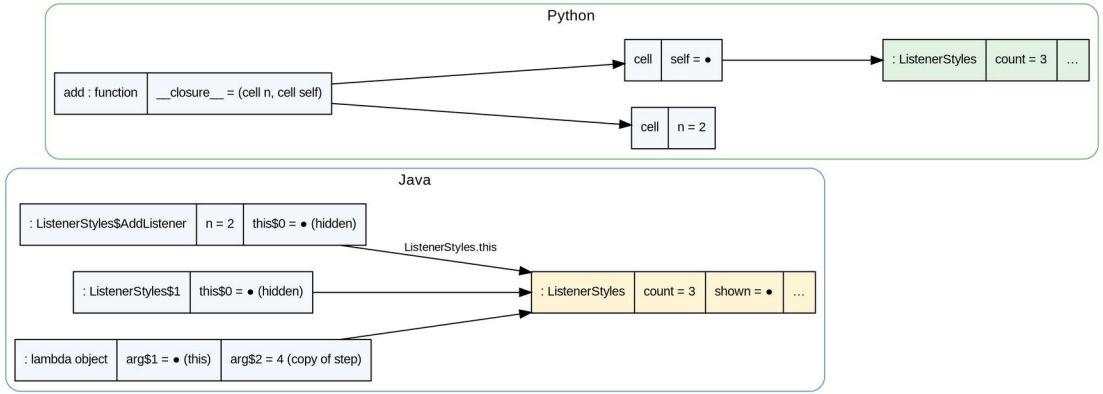


Figure 10.15 What listeners remember. Java inner and anonymous objects keep a hidden reference to their outer object; a lambda object holds copies of the values it captured. A Python closure keeps *cells* that refer to the variables of the function that created it. (The field names are the compilers' implementation details.)

Figure 10.15: inner and anonymous objects hold `this$0`; a lambda holds *copies*; a Python closure holds *cells*.

Book: Section 10.10.3 and Figure 10.15.

Effectively Final: Why Java Copies

A lambda may use a local variable only if it is **effectively final**: the lambda may run long after the method returned, so Java stores a *copy* — and forbids the variable from changing afterwards.

The natural loop `for (int n = 10; n <= 30; n += 10)` does not compile:

```
LoopCapture.java:9: error: local variables referenced from a lambda expression must be final or
effectively final
            b.addActionListener(e -> count += n);
                                              ^
1 error
```

`for (int n : new int[] {10, 20, 30})` is fine: a *new* `n` every pass. Fields are free: a lambda reaches `count` through the captured panel reference.

Book: Section 10.10.4.

Python: Five Callbacks and One Trap

A bound method, a closure (`make_adder`), a lambda, `functools.partial`, a callable object with `__call__` (Listing 10.11).

Listing 10.11 `listener_styles.py` (excerpt): the late-binding trap in a loop, and its cure with a default argument.

```
        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)))
```

The three “late” buttons all add 30: a closure stores the *variable* `n`, not its value, and reads it when **called** — **late binding**. `lambda n=n:` evaluates the default *now*.

```
late  +10 -> count = 30
late  +20 -> count = 60
late  +30 -> count = 90
fixed +10 -> count = 10
fixed +20 -> count = 30
fixed +30 -> count = 60
```

Book: Section 10.10.5 and Listing 10.14.

Drawing with the Mouse

One anonymous `MouseAdapter` serves as mouse *and* mouse-motion listener; the drawing lives in `strokes`, a `List<List<Point>>` (Listing 10.12).

Listing 10.12 `Doodle.java` (excerpt): an anonymous `MouseAdapter` that starts strokes, extends them and reports the pointer.

```
        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
```

Book: Section 10.10.6 and Listing 10.15.

Why Store the Points?

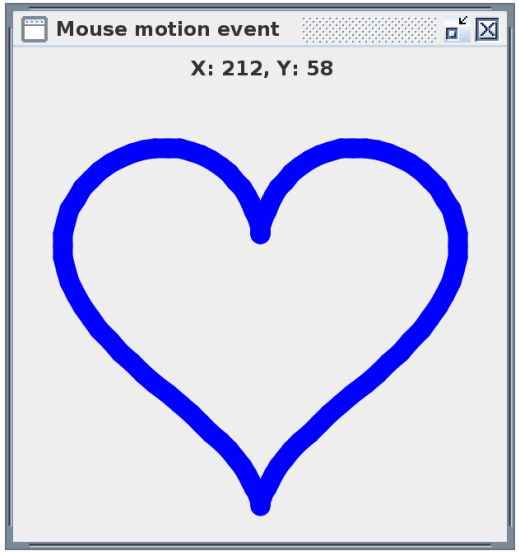


Figure 10.16 A heart drawn by dragging the mouse, with the coordinates of the pointer shown in the label.

Figure 10.16 can be drawn with `getGraphics()` inside `mouseDragged` — until the window is minimized or covered: then it is gone. Only what `paintComponent` draws from fields survives. In Tkinter each segment is a line item, and Tk keeps it (book Listing 10.16).

Book: Section 10.10.6 and Figure 10.16.

The Thread Rule

> **DEFINITION. The thread rule.** Swing components and Tkinter widgets may be created, read and changed only on the GUI thread — the event dispatch thread (EDT) in Swing, the thread running `mainloop()` in Tkinter. Conversely, the GUI thread must never be kept busy for long.

Everything GUI on the GUI thread: listeners already run there; other code uses `SwingUtilities.invokeLater(r)` or `invokeAndWait(r)` — which is why `main` builds the window with `invokeLater`.

Nothing slow on the GUI thread: about 0.1 s feels instantaneous; after about 1 s the user's flow of thought is interrupted.

Book: Section 10.11.1.

Slow Work, Wrong and Right

A month of orders, 20 ms per day, about 600 ms in all; the total both buttons must show is Equation (10.4):

$$\sum_{d=1}^{30} \left(100{,}000 + 5{,}000\left(d \bmod 7\right)\right) = 30 \times 100{,}000 + 5{,}000 \times 87 = 3{,}435{,}000 \tag{10.4}$$

Remainders $d \bmod 7$ for $d = 1, \ldots, 30$: four weeks of 21, plus $1+2$ — so $4 \times 21 + 3 = 87$.

The right button hands the tally to a `SwingWorker` (Listing 10.13):

Listing 10.13 `SlowWork.java` (excerpt): the `SwingWorker` — `doInBackground` on a worker thread, `done` back on the EDT.

```
            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();
                }
            }.execute();
```

Book: Section 10.11.2, Equation (10.4) and Listing 10.17.

The Thread Rule on a Time Line

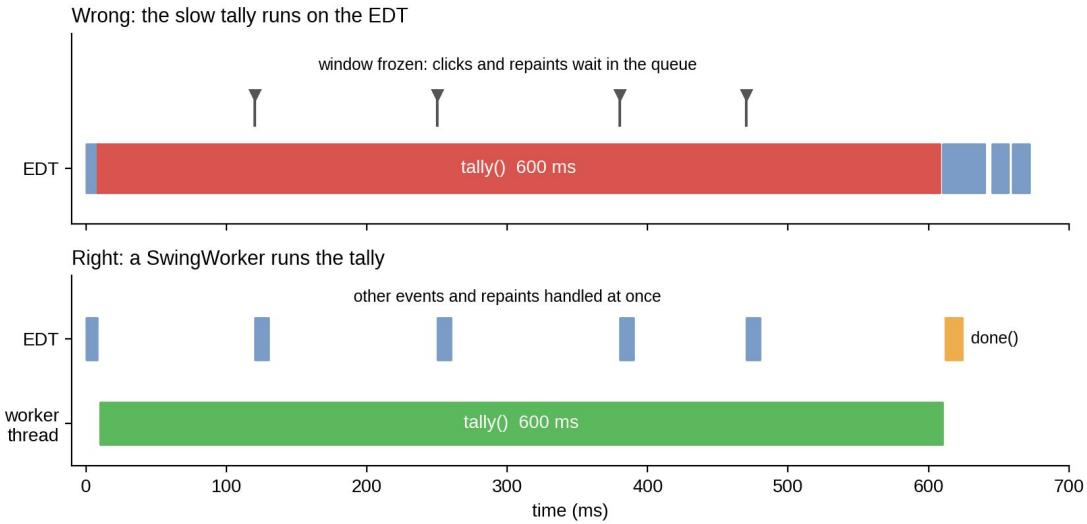


Figure 10.17 The thread rule on a time line. Top: the tally runs on the EDT, which cannot handle clicks or repaints for 600 ms. Bottom: a worker thread runs the tally while the EDT keeps handling events; `done()` returns to the EDT to show the result.

Figure 10.17: the log names the threads — `AWT-EventQueue-0` blocked 600 ms for the wrong button; `SwingWorker-pool-1-thread-1` for the right one, and `done()` back on the EDT.

Book: Section 10.11.2 and Figure 10.17.

The Same Recipe in Tkinter

Start a thread for the work; let it put its result into a thread-safe queue; let the GUI thread poll the queue with a timer (Listing 10.14).

Listing 10.14 `slow_work.py` (excerpt): a daemon worker thread, a `queue.Queue`, and `after` polling on the GUI thread.

```
    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())

    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)
```

`work` touches no widget. `poll` runs on `MainThread` every 50 ms until the result arrives.

Book: Section 10.11.2 and Listing 10.18.

Profitina Laundry: the Order-Entry Window

Form: name, phone, service (Wash & Fold, Iron Only, Wash, Dry & Iron), weight, *Express*.

Live price as soon as weight, service or express changes — before anything is added.

Validation: name up to 40 characters; phone 10–13 digits starting with 0; weight above 0 and at most 50 kg, and `4,2` accepted as well as `4.2`. Errors in red, cursor in the field to fix.

List, remove and total: ORD-0001, ORD-0002, …; *Remove selected* enabled only while a row is selected; the footer shows count and total.

Book: Section 10.12.

The Swing Window

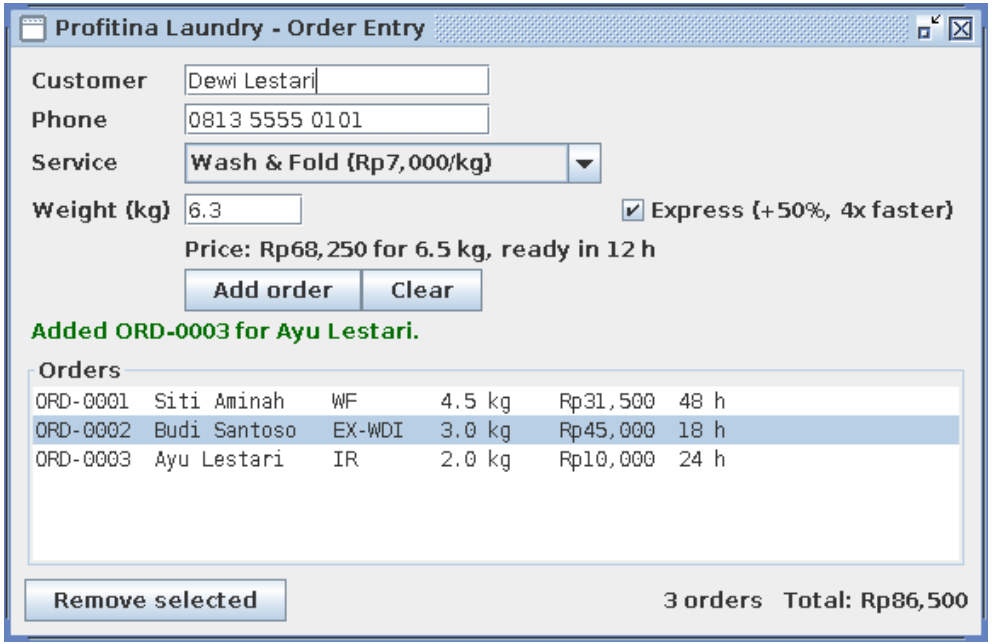


Figure 10.18 The Swing order-entry window with three orders, a fourth being typed (the live price already shows Rp68,250), and the second row selected.

Figure 10.18: three orders, a fourth being typed (the live price already shows Rp68,250), the second row selected.

Book: Section 10.12.1 and Figure 10.18.

The Tkinter Window

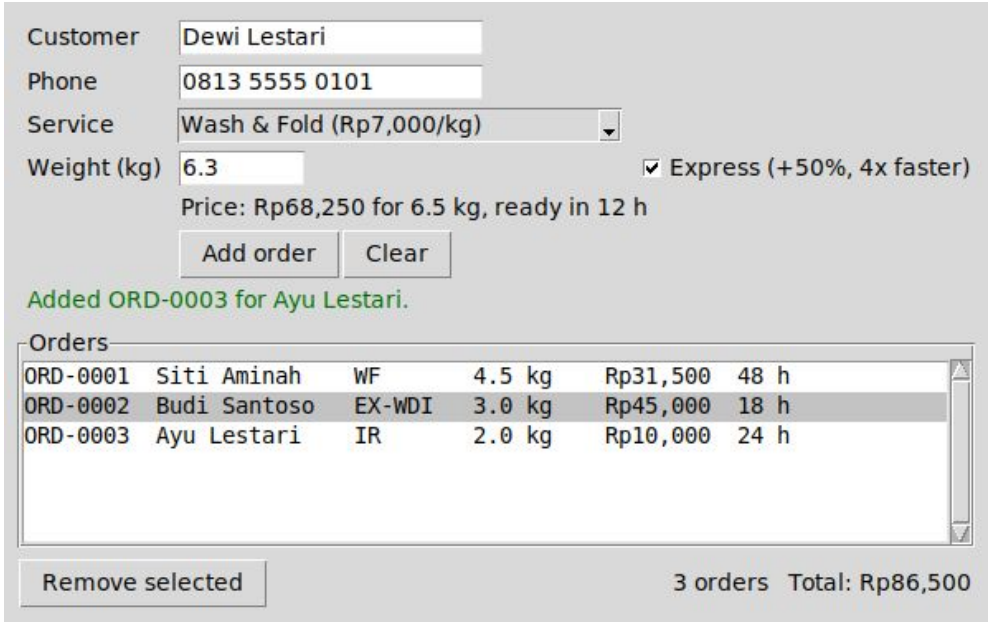


Figure 10.19 The Tkinter order-entry window in the same state. (The screenshot shows the window's contents without the title bar that the operating system adds.)

Figure 10.19: the same state — and the self-tests of the two versions print *identical* lines.

Book: Section 10.12.1 and Figure 10.19.

Worked Example: the Prices, Precisely

Chargeable weight Equation (10.5), price and turnaround Equation (10.6), express rule Equation (10.7):

$$\hat{w} = \frac{\lceil 2 \max(w, w_{\min}) \rceil}{2} \tag{10.5}$$

$$P = r \cdot \hat{w}, H = h \tag{10.6}$$

$$P_{\text{ex}} = \lfloor \frac{3P}{2} \rfloor, H_{\text{ex}} = \max\left(6, \lfloor \frac{h}{4} \rfloor\right) \tag{10.7}$$

Siti, 4.2 kg Wash & Fold: $\lceil 8.4 \rceil = 9$, $\hat{w} = 4.5$, $P = 31{,}500$, 48 h. **Budi**, 2.6 kg WDI express: $\hat{w} = 3.0$, $P = 30{,}000$, $P_{\text{ex}} = 45{,}000$, $H_{\text{ex}} = \max(6, 18) = 18$ h. **Ayu**, 1.3 kg ironing: $5000 \times 2 = 10{,}000$. Total $86{,}500$.

Dewi, 6.3 kg Wash & Fold express: $\lceil 12.6 \rceil = 13$, $\hat{w} = 6.5$, $P = 45{,}500$, $P_{\text{ex}} = 68{,}250$, 12 h — the live price of Figure 10.18.

Book: Section 10.12.2 and Equations (10.5)–(10.7).

A Model That Knows Nothing About Windows

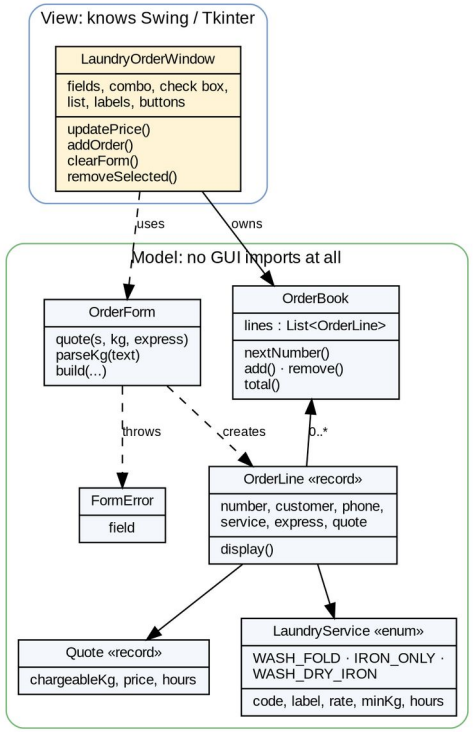


Figure 10.20 The design of the order-entry program. The view owns an `OrderBook` and uses `OrderForm` for every calculation and check; the model classes never mention a window.

Figure 10.20: `LaundryService`, `Quote`, `OrderLine`, `FormError`, `OrderForm`, `OrderBook` import nothing from Swing or Tkinter; the view asks the model for every decision.

Book: Section 10.12.3 and Figure 10.20.

The Rules in the Model

`OrderForm.quote` is Equation (10.5)–Equation (10.7) in code; the 10^{-9} guards against binary rounding (Listing 10.15).

Listing 10.15 `LaundryOrderWindow.java` (excerpt): `OrderForm.quote`, the pricing rules in integer arithmetic.

```
    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);
    }
```

`parseKg` turns `4,2` into 4.2 or throws a `FormError` that names the field to fix.

Book: Sections 10.12.2–10.12.3 and Listing 10.19.

Connecting the Events

All the wiring in one method, one line per reaction (Listing 10.16):

Listing 10.16 `LaundryOrderWindow.java` (excerpt): `wireEvents`, one line per reaction.

```
    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());
    }
```

A `DocumentListener` hears every keystroke (an action listener would fire only on Enter). Tkinter: `weight_var.trace_add("write", ...)` (book Listing 10.20).

Book: Sections 10.12.4 and 10.12.6, Listing 10.19.

The Self-Test of the Whole Window

```
   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
```

Requirements checked one by one: bad phone and three bad weights rejected; three orders, total Rp86,500; after removing ORD-0002, $86,500 - 45,000 = 41,500$ and *Remove selected* disabled again.

Book: Section 10.12.5.

Profitina in Practice

Profitina's products are web applications, but the browser runs exactly this model: one GUI thread, `addEventListener`, an event loop; a slow listener freezes the page.

Optimistic posting is the thread rule in action: the composer clears at once, the request runs in the background, and failure restores the text — the shape of `SwingWorker`.

The live feed arrives by Server-Sent Events: the page registers a listener and waits — Pseudocode 10.1 with the server as one more source of events.

The dashboard is a view over a model; the rules live on the server, testable without a browser. The window advises; the person at the counter presses *Add order*.

Book: Section 10.13.

Common Mistakes

Slow work in a listener; touching components from another thread; forgetting `EXIT_ON_CLOSE`; `setVisible(true)` before adding components.

Drawing outside `paintComponent`; forgetting `super.paintComponent(g)`; calling `paintComponent` yourself instead of `repaint()`.

Expecting a listener to run where it is registered; the late-binding trap; `command=self.add_order()` — calling the callback instead of passing it.

Mixing `pack` and `grid` in one parent; absolute positions; rules inside the listeners instead of in a model.

Book: Section 10.14.

Summary: Five Things to Remember

1. A GUI program is event-driven: build the window, register handlers, hand control to the event loop — inversion of control.

2. A window is a tree of components; layout managers and `pack`/`grid` place them by rules; nest panels for complex windows.

3. Paint from state: `paintComponent` + `repaint()` in Swing, canvas items in Tk.

4. Inner and anonymous classes keep the outer object; lambdas copy effectively final locals; Python closures bind late — fix loops with `lambda n=n:` or `partial`.

5. The thread rule: touch the GUI only on its thread, never block it; keep the rules in a toolkit-free model with a self-test.

Book: Section 10.15.

Next Time

Chapter 11 — The Model–View–Controller Pattern. Today's split between a model and a view becomes a pattern with three roles.

The Observer pattern underneath MVC, the Up/Down counter rebuilt, a predictive phone keypad, and the Profitina Laundry front desk as a complete MVC application.

Before the meeting: Exercises 10.1, 10.2, 10.8 and 10.9, and read Sections 11.1 to 11.4.

Book: Section 10.16 and Chapter 11.

Chapter 1

Lecture 11 — The Model–View–Controller Pattern

Object-Oriented Programming (EF234302)

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*

Lecturer: Irfan Subakti — yifana@gmail.com

Book: Chapter 11, The Model–View–Controller Pattern. Every page of this handout ends with a pointer into the book, which is the primary reference; the handout is the chapter in miniature.

Today's Meeting

1. The front desk that knew too much, and the three roles: model, view, controller — and who may know whom (Sections 11.2–11.3).

2. The smallest MVC program, and the Observer pattern underneath MVC in Java and Python (Sections 11.4–11.5).

3. Two windowed MVC programs: the Up/Down counter and a predictive phone keypad (Sections 11.6–11.7).

4. Testing the model without the GUI (Sections 11.8 and 11.10).

5. The Profitina Laundry front desk as a complete MVC application; MVP and MVVM (Sections 11.9–11.13).

Book: Chapter 11, Sections 11.1 to 11.14.

Where We Left Off

Chapter 10 gave our programs windows: frames and panels, layout managers, and the central idea — after the window is built, the program *waits*; the toolkit calls our listeners.

Its programs were small, one class each: the class built the window, and its listeners did whatever the button was for. Fine for twenty lines; broken at two hundred.

Underneath today's pattern lies one we already used without the name: every `addActionListener` subscribed an **observer** to a button.

> **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 it (the views), and let a third object translate clicks into requests to the model (the controller).

Book: Section 11.1.

The Front Desk That Knew Too Much

One class, `FrontDeskWindow extends JFrame`; the *Add order* listener read the fields, checked the weight, computed the price, stored a table row and redrew the totals. Then, in one year:

A second screen for ready orders — but the orders lived only as `Object` arrays in the window's table.

New price rules (minimum weights) — hidden inside a button listener; a misplaced parenthesis was found by a customer.

Tests ("does 4.2 kg Wash & Fold still cost Rp31,500?") — impossible without a screen. **A web version** — nothing reusable.

Book: Section 11.2.

Before and After MVC

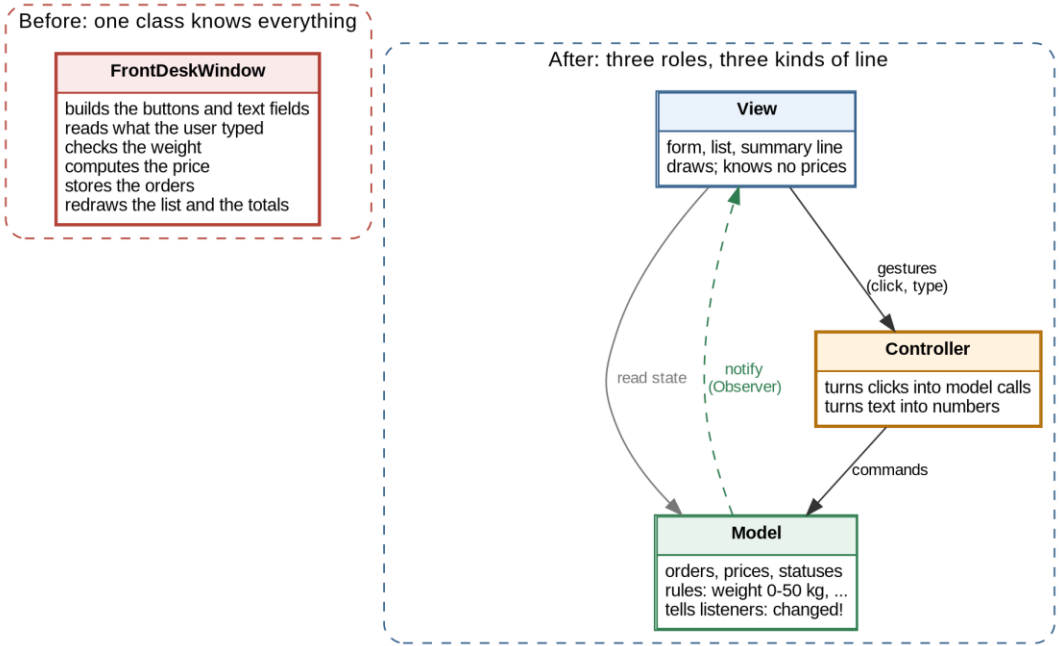


Figure 11.1 The same front desk before and after MVC. Left: one window class does six jobs. Right: the model holds the orders and the rules, the view draws, and the controller translates gestures into model calls; the dashed arrow is the Observer notification by which the model tells the view that something changed.

Figure 11.1: jobs that change for different reasons — the look, the business rules, the storage — were welded into one listener. After MVC, each of the four stories touches exactly one part.

Book: Section 11.2 and Figure 11.1.

Three Roles

> **DEFINITION.** The **model** represents the application's data and the rules that govern access to and updates of that data. The **view** renders the contents of a model: it decides how the data is presented, and when the data changes, it updates its presentation. The **controller** translates the user's interactions with the view into actions that the model performs.

The model's definition says nothing about pixels or buttons; the view's says nothing about what the data *means*.

The controller is a translator between the language of gestures ("*Add order* clicked while the weight field held `4,2`") and the language of the business ("add 4.2 kg of Wash & Fold for Sari").

Book: Section 11.3.

The Classic Diagram

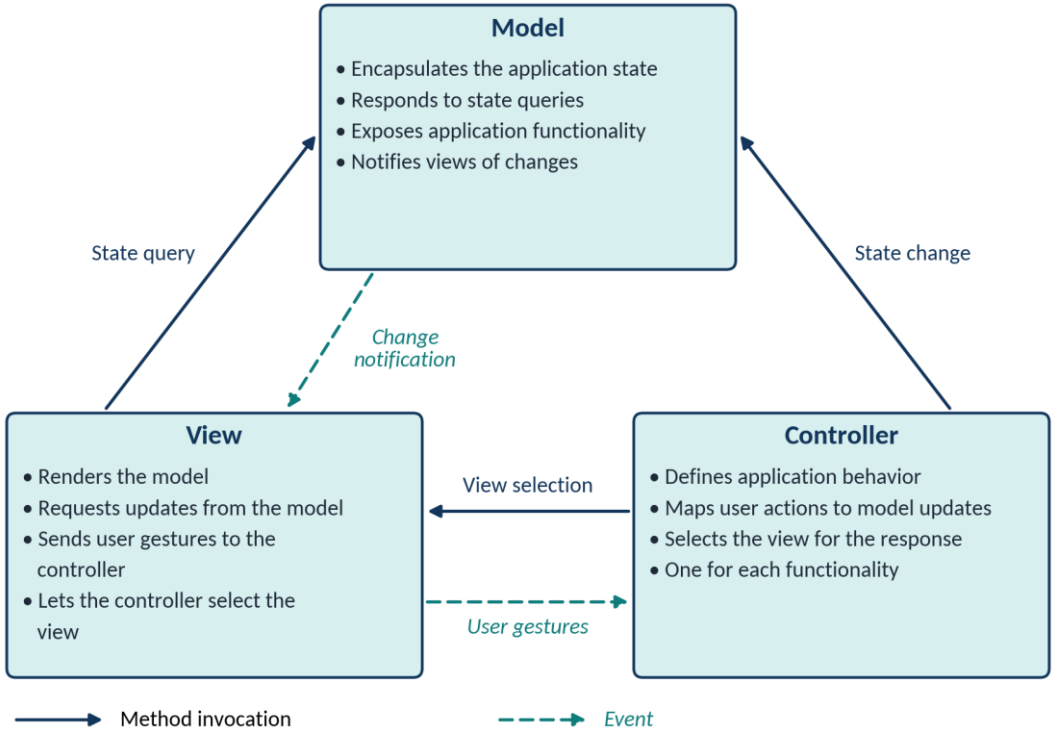


Figure 11.2 The classic MVC diagram. The model encapsulates the application state, answers state queries, exposes the application's functionality and notifies views of changes; the view renders the model and sends user gestures to the controller; the controller maps user actions to model updates and selects the view for the response.

Figure 11.2: solid arrows are method calls, dashed arrows are events. The model never calls a view by name — it only announces "I have changed".

Book: Section 11.3 and Figure 11.2.

Roles You Can Point At

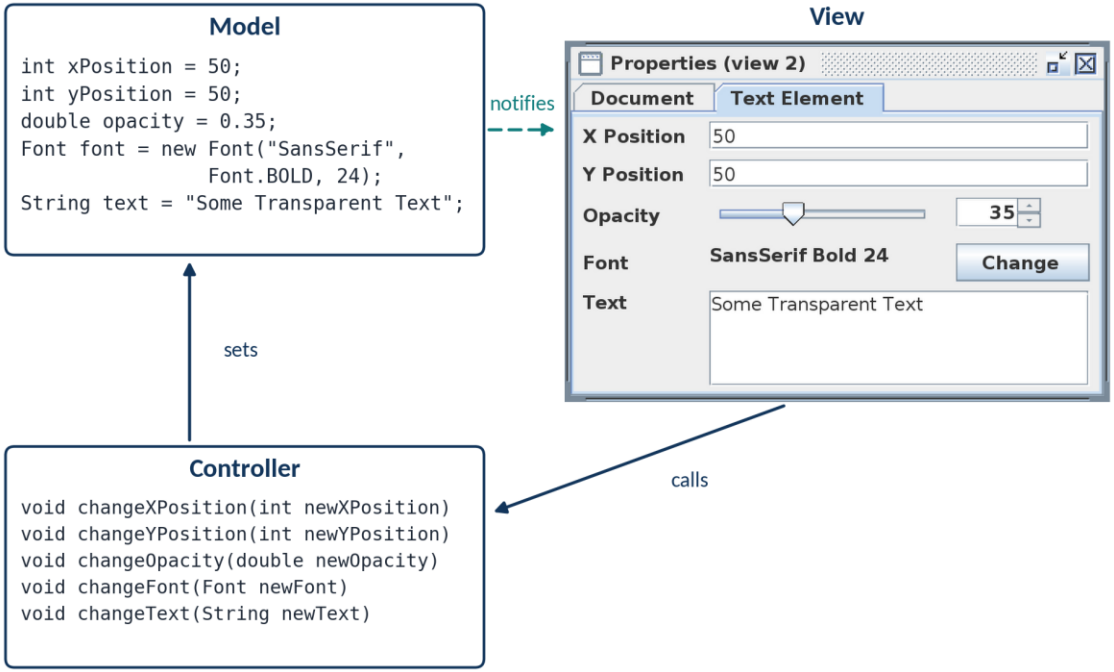


Figure 11.3 Model, view and controller of a small document editor. The model is five plain values; the view is a properties window that displays them; the controller offers one method per property, which the view calls when the user edits a field.

Figure 11.3: drag the opacity slider to 50 and the spinner changes too — *because the model changed*, not because the slider told it. Two widgets that show one value can never disagree.

Book: Section 11.3.1 and Figure 11.3.

One Model, Many Views

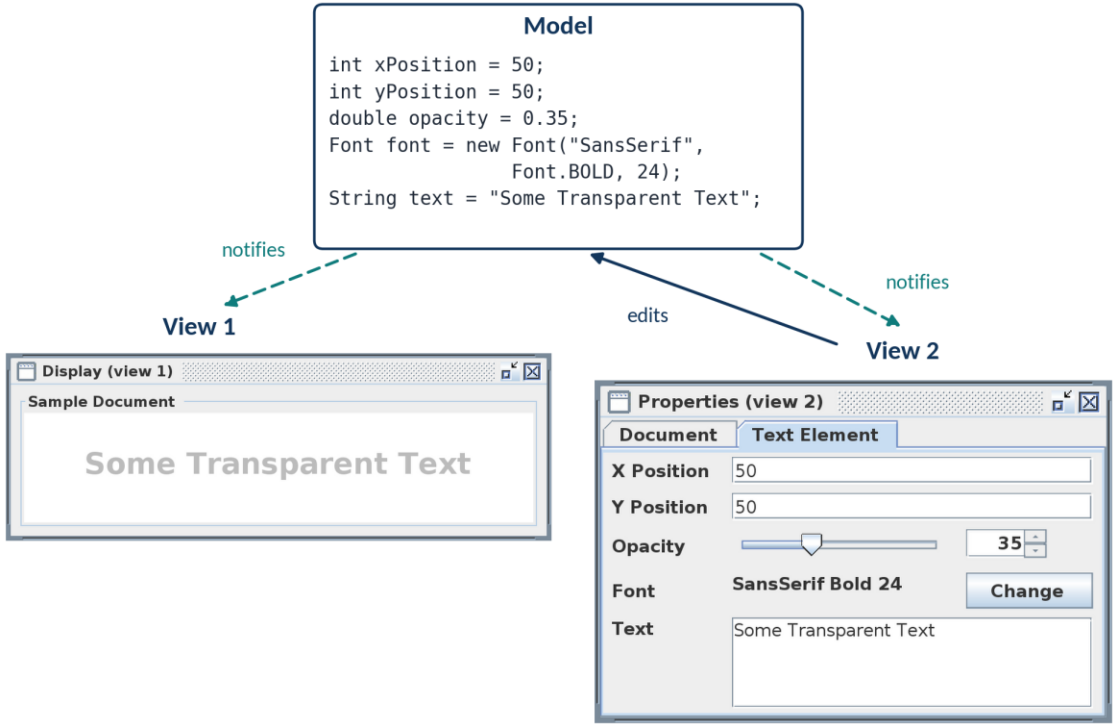


Figure 11.4 One model, two views. The *Display* window draws the text as it will appear on the page, and the *Properties* window shows the same values as numbers; both are kept current by the model's change notifications.

Figure 11.4: the *Display* and *Properties* windows do not know each other; each knows only the model.

Book: Section 11.3.1 and Figure 11.4.

A View Is a Function of the Model

At moment t the model is in state S_t; ideally a view is a render function R of that state, Equation (11.1):

$$V_t = R(S_t) \tag{11.1}$$

Two views are $R_1(S_t)$ and $R_2(S_t)$: they cannot contradict each other.

A test of the model needs no view: it checks S_t directly.

A text field that holds a value existing nowhere else violates Equation (11.1) — the place where two screens will one day disagree.

Book: Section 11.3.1 and Equation (11.1).

Who May Know Whom

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

Table 11.1, the most important cell: **the model must not know the GUI toolkit** — no `javax.swing`, no `java.awt`, no `tkinter`. Easy to check, and it guarantees testability, reuse and independence from redraw order.

Book: Section 11.3.2 and Table 11.1.

The Controller as a Mediator

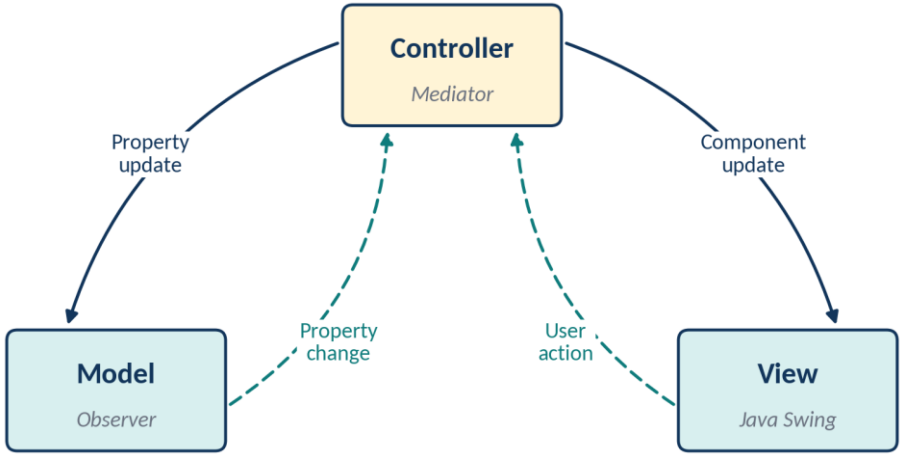


Figure 11.5 The controller as a mediator. User actions travel from the view to the controller, which updates the model's properties; the model's property changes travel back to the controller, which updates the view's components. The model and the view never refer to each other.

Figure 11.5: model and view never meet. The keypad lets its views observe the model; the laundry desk routes everything through a mediating controller.

Book: Section 11.3.2 and Figure 11.5.

Where the Idea Came From

1979: Trygve Reenskaug at Xerox PARC, working on Smalltalk — one user, the same information seen and changed in several ways at once.

Built into the Smalltalk-80 class library; the **Observer** pattern that connects models to views was cataloged in *Design Patterns* (1994).

Swing itself is MVC-flavored: every `JTable` has a `TableModel`, every `JList` a `ListModel`, every `JButton` a `ButtonModel`.

Today's controllers receive "button *Add order* clicked", not "mouse down at pixel (412, 37)": the toolkit does the raw input.

Book: Section 11.3.3.

The Smallest MVC Program

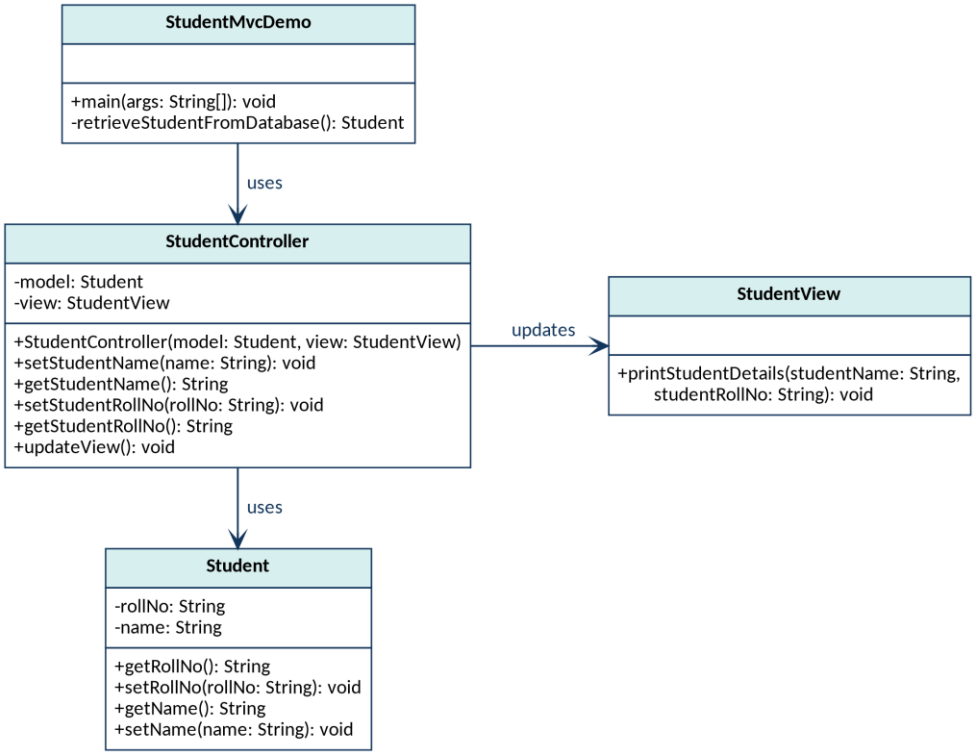


Figure 11.6 Class diagram of the student example. `StudentController` holds a `Student` model and a `StudentView`; it updates the model through setters and updates the view with `updateView()`; the demo class `StudentMvcDemo` uses the controller and retrieves the first record from a (simulated) database.

Figure 11.6: `Student` has no arrow to anything — the model knows nobody. No GUI at all: MVC is about *responsibilities*, not about Swing.

Book: Section 11.4 and Figure 11.6.

The Controller Knows Both

Listing 11.1: the only class that holds the model *and* the view, in `final` fields.

Listing 11.1 `StudentMvcDemo.java` (excerpt): `StudentController`, built from a model and a view.

```
// 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);
    }
```

`updateView()` asks the model for its values and hands them to the view.

Book: Section 11.4 and Listing 11.1.

Four Steps, and a Stale View

Every MVC `main`: **build the model, build the view, build the controller with both, let the controller drive (Listing 11.2).**

Listing 11.2 `StudentMvcDemo.java` (excerpt): `main`, the four steps.

```
    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();
    }
```

After `setStudentName("John")` the model says *John* while the screen still says *Robert* — until somebody remembers `updateView()`. In a real program, somebody forgets.

Book: Section 11.4 and Listing 11.1.

The Observer Pattern

> **DEFINITION.** In the **Observer** pattern, a **subject** keeps a list of **observers**. Observers register by calling a method such as `addListener`, and may unregister. Whenever the subject's state changes, it **notifies** every registered observer by calling one agreed method on each. The subject knows only that its observers have that method.

A `JButton` is a subject of `ActionListener`s; a Tkinter `StringVar` is a subject of `trace_add` callbacks; a newsletter is a subject of its subscribers.

Subject: keep a list, offer *add* and *remove*, call everybody on change. Observer: have the method.

Book: Section 11.5.

The Observer Pattern in Java

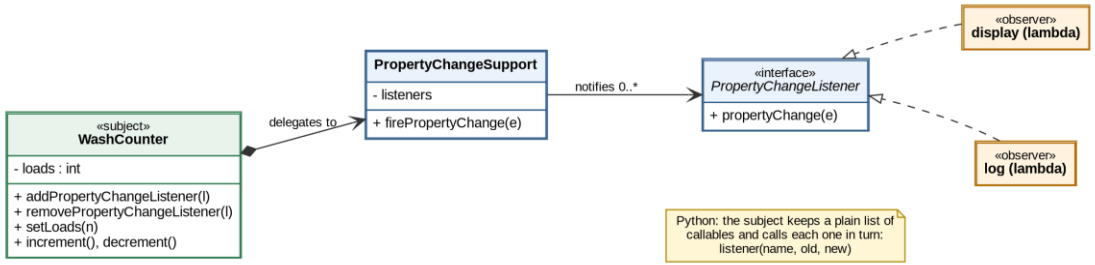


Figure 11.7 The Observer pattern as used by `ObserverDemo.java`. The subject `WashCounter` delegates its list of listeners to a `PropertyChangeSupport`, which notifies any number of objects implementing `PropertyChangeListener`; the two observers are lambdas. In Python the same roles are played by a plain list of callables.

Figure 11.7: `WashCounter` delegates its listener list to `java.beans.PropertyChangeSupport`; the observers are two lambdas.

Book: Section 11.5 and Figure 11.7.

From Observable to PropertyChangeSupport

`java.util.Observable` is **deprecated** since Java 9: a *class* (steals your single inheritance), a bare `Object` argument, no property name.

`PropertyChangeSupport` is used by composition, sends name, old and new value — and fires **no event when nothing changed** (**Listing 11.3**).

Listing 11.3 `ObserverDemo.java` (excerpt): `setLoads` validates, stores and fires; `increment` and `decrement` go through it.

```
    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);
    }
```

Book: Section 11.5.1 and Listing 11.3.

Worked Example: Reading the Notifications

```
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
```

Each `increment` reaches `display` then `log`, in registration order; the counter itself prints nothing. `setLoads(2)` prints nothing: old and new are both 2. After `removePropertyChangeListener(log)`, only the display hears. `setLoads(-1)` is refused *before* anything changes or anyone is told.

Book: Section 11.5.1 and Listing 11.3.

Python: an Observer in a Dozen Lines

A function is an object, so the subject keeps a list of *functions* and calls each (**Listing 11.4**).

Listing 11.4 `observer_demo.py` (excerpt): validate, store, fire only on a real change — over a copy of the list.

```
    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)
```

`list(self._listeners)`: a listener may unsubscribe itself while being told, and removing from a list during a `for` loop silently skips the next one.

Book: Section 11.5.2 and Listing 11.4.

The Pattern in Two Languages

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)`

Table 11.2: what `PropertyChangeSupport` gives Java for free, Python writes in one line each.

Book: Section 11.5.2 and Table 11.2.

One Change, Many Calls

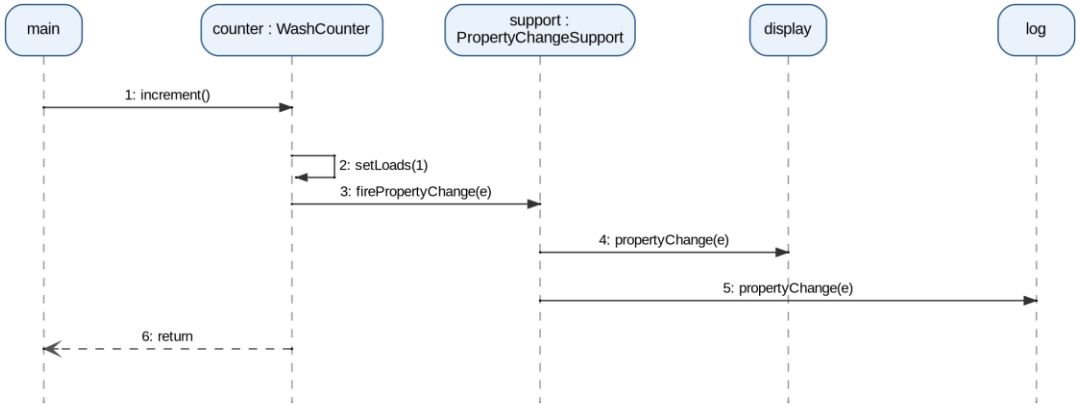


Figure 11.8 Sequence diagram of one `increment()`. The counter calls its own `setLoads(1)`, which asks the support object to fire one event; the support object calls each listener in registration order, and only then does `increment()` return to `main`.

Figure 11.8: notification is **synchronous** — messages 4 and 5 happen *inside* message 1. With k observers and m changes: Equation (11.2).

$$C = m \cdot k \tag{11.2}$$

Book: Section 11.5.3, Figure 11.8 and Equation (11.2).

Push or Pull?

Push: every event carries the new (and old) value; observers react without asking. Convenient when they need exactly that value.

Pull: the event says only "I changed"; each observer queries the subject for what it needs. Simpler for large models with different views, and it cannot deliver stale data.

A model should notify once per logical change: three events for one click triple m in Equation (11.2) for nothing.

The counter pushes a single integer; the keypad and the laundry desk pull.

Book: Sections 11.5.3–11.5.4.

The Counter, Rebuilt

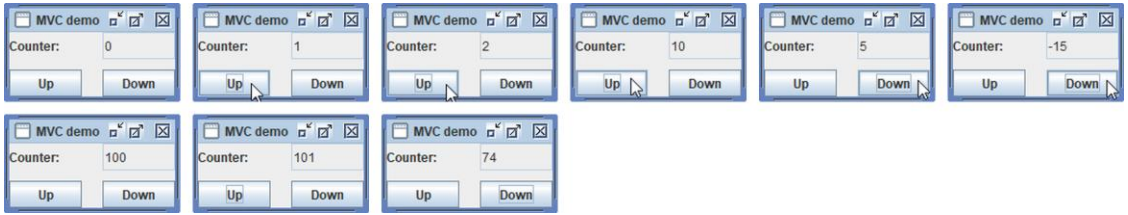


Figure 11.9 The Up/Down counter at work. Top row, started at 0: after clicks on *Up* the field shows 1, 2 and 10; after clicks on *Down* it shows 5 and −15. Bottom row, started at 100: 101, then 74. The mouse pointer shows which button was clicked last.

Figure 11.9: the original classes put the controller's work in a class called `View`, left `Controller` almost empty, and extended the deprecated `Observable`. The rebuilt version keeps the window and moves each job home.

Book: Section 11.6 and Figure 11.9.

The Model Pushes an Integer

`CounterModel` keeps `IntConsumer` observers and calls `accept(value)` on every change; `increment` and `decrement` go through `setValue` (**Listing 11.5**).

Listing 11.5 `CounterMvc.java` (excerpt): the counter model — one place where the value changes, one place where observers are told.

```
    void addListener(IntConsumer listener) {
        listeners.add(listener);
    }

    void increment() {
        setValue(value + 1);
    }

    void decrement() {
        setValue(value - 1);
    }

    void setValue(int newValue) {
        value = newValue;
        for (IntConsumer listener : listeners) {
            listener.accept(value);  // notify every observer
        }
    }
```

No Swing in sight: the model has no idea whether it is shown in a window, printed, or not shown at all.

Book: Section 11.6.1 and Listing 11.5.

A View Contract and a Four-Line Controller

Listing 11.6 `CounterMvc.java` (excerpt): the `CounterView` interface and the controller that wires model and view.

```
// 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
    }
}
```

Listing 11.6: `Runnable` is "code that takes nothing, returns nothing", so `model::increment` fits. The last line draws **the first picture** — forget it and the window opens empty.

Book: Section 11.6.1 and Listing 11.5.

The Self-Test: a View That Prints

The controller sees the view only through `CounterView`, so a `TextCounterView` that prints can replace the window:

```
  [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
```

Proven: the model counts, *Up* means increment, the observer is registered, the first picture is drawn. **Not** proven: that the window looks right — so keep the view too thin to get wrong.

Python (book Listing 11.6): duck typing replaces the interface, and `import tkinter` sits *inside* the view's constructor so the self-test runs even without Tkinter.

Book: Sections 11.6 and 11.6.3, Listing 11.5.

Words, Keys and Signatures

Before touch screens: keys 2–9 carry letters; **T9** guesses the word from one press per letter. *hello* is 4-3-5-5-6.

With κ mapping a letter to its key, the **signature** is Equation (11.3):

$$\sigma(c_1 c_2 \ldots c_n) = \kappa(c_1)\,\kappa(c_2) \ldots \kappa(c_n) \tag{11.3}$$

$\sigma(\text{hello}) = 43556$, $\sigma(\text{wash}) = 9274$, $\sigma(\text{good}) = \sigma(\text{home}) = 4663$. σ is not one-to-one — hence the star key. A map from signature to sorted words makes each lookup one map access.

Book: Section 11.7.1 and Equation (11.3).

What Each Key Does

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

Table 11.3: when the signature matches no word — `435` in the middle of *hello* — the model shows the digits themselves.

Book: Section 11.7.1 and Table 11.3.

The Keypad's Classes

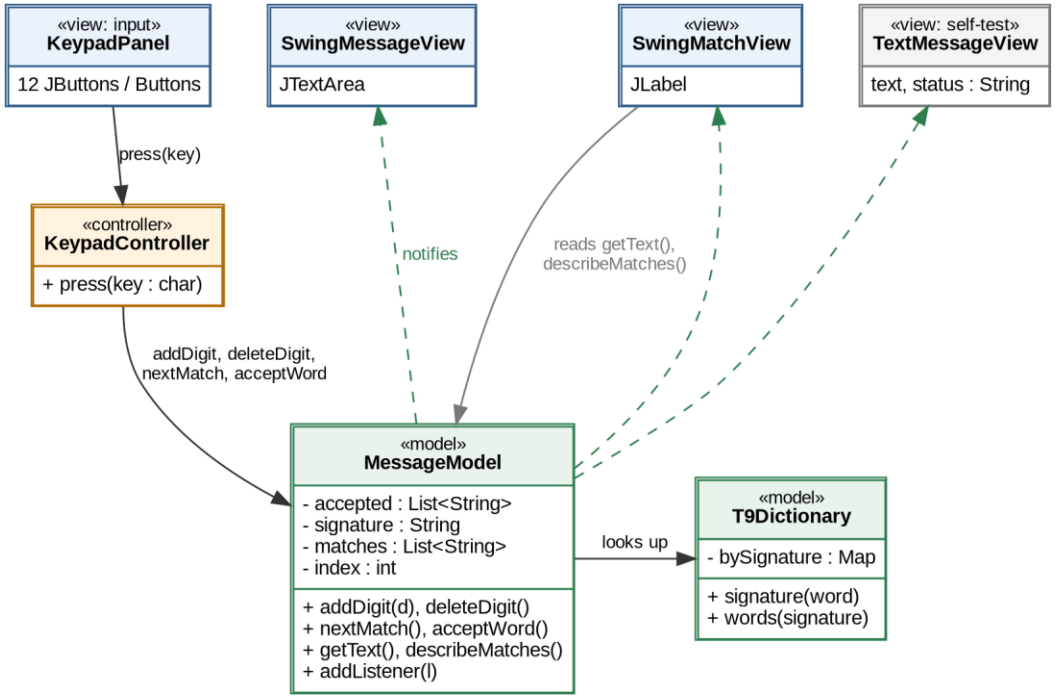


Figure 11.10 The classes of the predictive keypad. The keypad panel reports each press to the controller, which calls one of the model's four operations; the model uses the dictionary and notifies its three views, which read the text and the current match back from the model.

Figure 11.10: the panel reports presses to the controller; the model uses the dictionary and notifies three views, which pull what they need.

Book: Section 11.7.1 and Figure 11.10.

The Model Protects Its Invariant

Each operation checks whether it has anything to do, changes the state, and fires — once (**Listing 11.7**).

Listing 11.7 `PredictiveKeypad.java` (excerpt): `addDigit` refuses anything but 2–9; `deleteDigit` fires only on a real change.

```
    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();
        }
    }
```

Book: Section 11.7.2 and Listing 11.7.

The Controller Is a Table

Listing 11.8 is **Table 11.3** written as a `switch`: that key 0 means "accept the word" is a fact about *this keypad*, not about messages.

Listing 11.8 `PredictiveKeypad.java` (excerpt): `KeypadController.press`.

```
// Controller: translates key presses into model operations.
class KeypadController {
    private final MessageModel model;

    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);
        }
    }
```

The button labels come from the same `LETTERS` table as the lookup, so they cannot disagree (in the original, key 9 said *vwxy*).

Book: Section 11.7.2 and Listing 11.7.

Worked Example: the Self-Test as a Film

```
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)
```

`4`: only *i*. `43`: *he* and *if*, alphabetically *he* first. `435`, `4355`: no word, digits shown. `43556`: *hello*. `4663` gives *gone*, two stars give *good*, then *home* (3 of 6).

Book: Section 11.7.4.

The Same Keypad in Python

Model and controller are line-for-line translations; `TkKeypadWindow` holds both views and the twelve buttons (book Listing 11.8).

Each button's command is `lambda k=key: controller.press(k)`. Without `k=key`, all twelve lambdas share the loop variable, which ends as `"C"` — every button would delete a digit.

The `Text` widget is switched to `"normal"` to change it and back to `"disabled"` so the user cannot type into it.

`python3 predictive_keypad.py --selftest` prints output identical to Java's, character for character.

Book: Section 11.7.5.

Testing the Model Without the GUI

A **unit test** checks one behavior in isolation, has a descriptive name, and reports which behavior failed. The keypad model imports no GUI, so it is tested like any class.

Test every *promise*, including what the model will *not* do: one event per real change, none for a change that is not one (**Listing 11.9**).

Listing 11.9 `test_keypad_model.py` (excerpt): one event per real change, and a refused key tested with `assertRaises`.

```
    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")
```

KEY INSIGHT. “How do I test my GUI?” is usually the wrong question. Ask “how little logic can I leave in my GUI?”

Book: Section 11.8 and Listing 11.10.

The Front Desk: Requirements

Name, service, weight, *Express*; **Add order** records the next number and shows the price.

Refused, changing nothing: empty name, weight not a number, ≤ 0 or above 50 kg; `2,6` means 2.6.

The list shows number, customer, service, weight, express, price, status; **Advance status** moves the selected order one step: *Received*, *Washing*, *Ready*, *Picked up*.

A summary line: orders, how many *Received*, how many *Ready*, total revenue.

Book: Section 11.9.

Worked Example: the Prices by Hand

Chargeable weight Equation (11.4), base price Equation (11.5), express Equation (11.6):

$$\widehat{w} = \frac{\lceil 2\max(w, w_s) \rceil}{2} \tag{11.4}$$

$$p = r_s \cdot \widehat{w} \tag{11.5}$$

$$p_{\text{ex}} = \lfloor \frac{3p}{2} \rfloor \tag{11.6}$$

Sari, 4.2 kg Wash & Fold: $\max(4.2,3) = 4.2$, $\lceil 8.4 \rceil = 9$, 4.5 kg, $p = 31{,}500$. **Budi**, 2.6 kg WDI express: 3 kg, $p = 30{,}000$, $p_{\text{ex}} = 45{,}000$. **Citra**, 1.3 kg ironing: 2 kg, 10,000. Total **Rp86,500**.

Book: Section 11.9.1 and Equations (11.4)–(11.6).

The Life of an Order

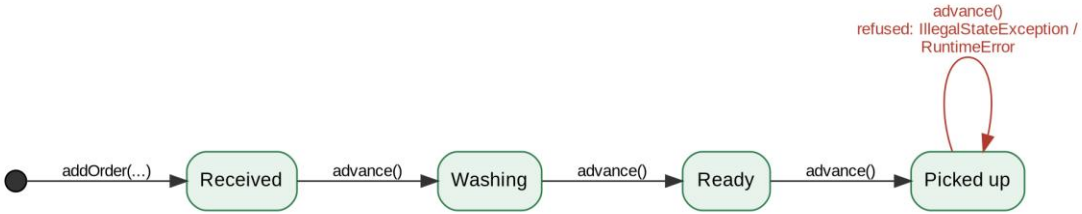


Figure 11.11 The life of a laundry order. Each `advance()` moves an order one state to the right; an order that has been picked up refuses to advance.

Figure 11.11: a small **state machine**. Each `advance()` moves one state to the right; *Picked up* refuses.

Book: Section 11.9.1 and Figure 11.11.

The Design of the Front Desk

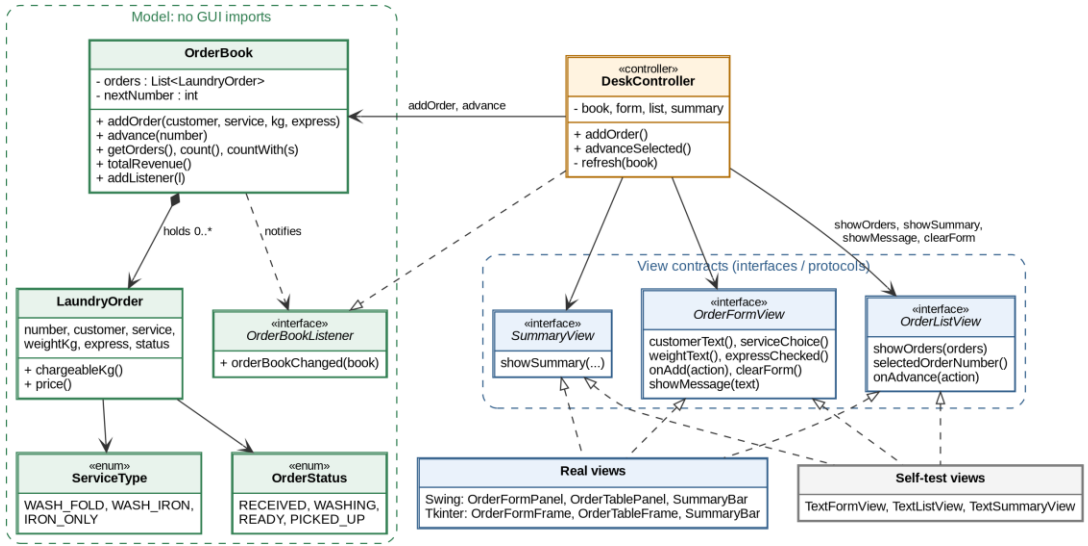


Figure 11.12 Class diagram of the front desk. The model (left) imports nothing from the GUI. The controller implements `OrderBookListener`, calls the model and updates three view contracts; the real Swing or Tkinter views and the text views of the self-test both implement those contracts.

Figure 11.12: one small view contract per part of the window — the waiting-room television needs a list and nothing else.

Book: Section 11.9.2 and Figure 11.12.

Validate, Change, Notify, Return

Listing 11.10: every refusal leaves *before* anything changes — no order number used, no view redrawn.

Listing 11.10 `OrderBook.java` (excerpt): `addOrder` validates, changes, notifies once, returns.

```
    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;
    }
```

`!(weightKg > 0 && weightKg <= MAX_KG)` also rejects `NaN`, which makes *every* comparison false.

Book: Section 11.9.3 and Listing 11.11.

Two Exceptions, Two Meanings

Listing 11.11 `OrderBook.java` (excerpt): `advance` refuses a picked-up order.

```
    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();
    }
```

Listing 11.11: `IllegalArgumentException` — "you asked about something that does not exist"; `IllegalStateException` — "the request is fine, but not in this state".

`getOrders` returns an unmodifiable view; `fireChanged` loops over `List.copyOf(listeners)`. Python: `orders()` returns a tuple, `advance` raises `RuntimeError`.

Book: Section 11.9.3 and Listings 11.11 and 11.12.

The Controller Turns Text into Values

Listing 11.12 `LaundryDesk.java` (excerpt): the controller parses the weight; the model judges it.

```
    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 11.12: `abc` is not a weight at all, so the model never sees it; whether 75 kg is acceptable is a *business* rule, and the model decides.

Book: Section 11.9.4 and Listing 11.13.

The Self-Test: a Shift at the Desk

```
> 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
```

The refused orders print only a message: the model did not change, no view redrew, and no number was used — Citra's order is #3. The total is the Rp86,500 we computed by hand.

Book: Section 11.9.4.

One Click on Add Order

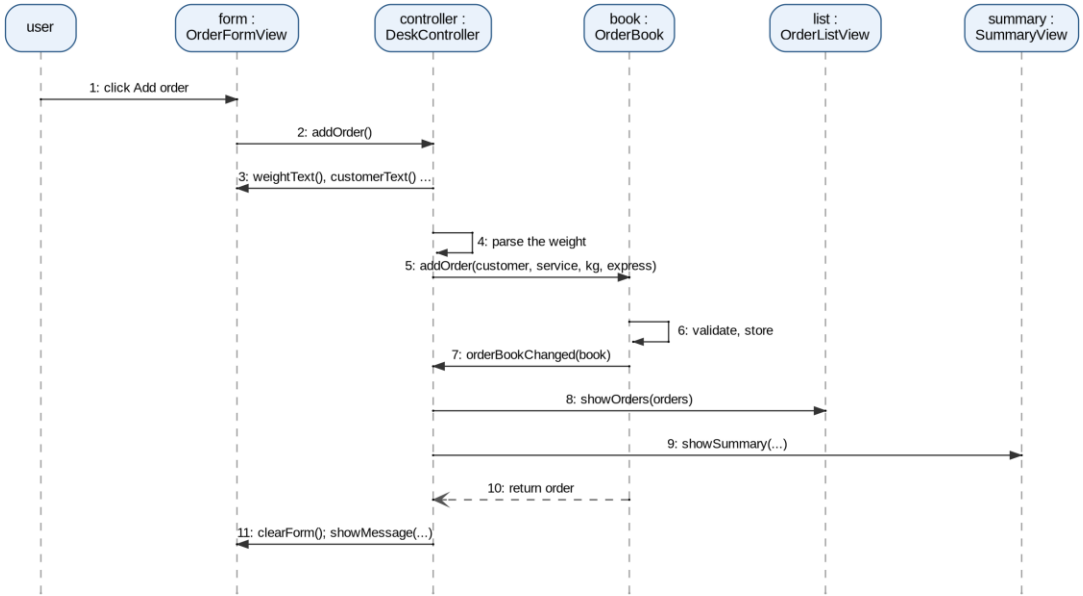


Figure 11.13 Sequence diagram of a click on *Add order*. The form reports the click to the controller, which reads the form, parses the weight and calls the model; the model validates, stores and notifies the controller, which refreshes the list and the summary; after the model returns, the controller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

Figure 11.13: messages 7–9 (notify, redraw) happen *inside* message 5, before the model returns; message 11 updates the form.

Book: Section 11.9.4 and Figure 11.13.

The Swing and Tkinter Views

No `if` about business in the whole view file: every method builds widgets or moves data between a widget and a parameter (book Listing 11.14).

A lesson the self-test could not see: all eight form components in *one* `FlowLayout` row — the `NORTH` region gave one line's height, and **Add order** wrapped out of sight. Only looking at the window revealed it.

Python: the view contracts are `typing.Protocol`s — duck typing written down; the Tkinter module is imported only when a window opens (book Listings 11.15 and 11.16).

Book: Sections 11.9.5–11.9.6.

Unit Tests for the Front Desk

`setUp` builds the same three orders before *every* test; `subTest` reports each refused case by name (**Listing 11.13**).

Listing 11.13 `test_order_book.py` (excerpt): five refusals in one loop with `subTest`.

```
    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)
```

The listener is the bound method `self.count_event`, so `remove_listener` receives an *equal* object — two lambdas with the same body are two different objects.

Book: Section 11.10 and Listing 11.18.

MVP and MVVM

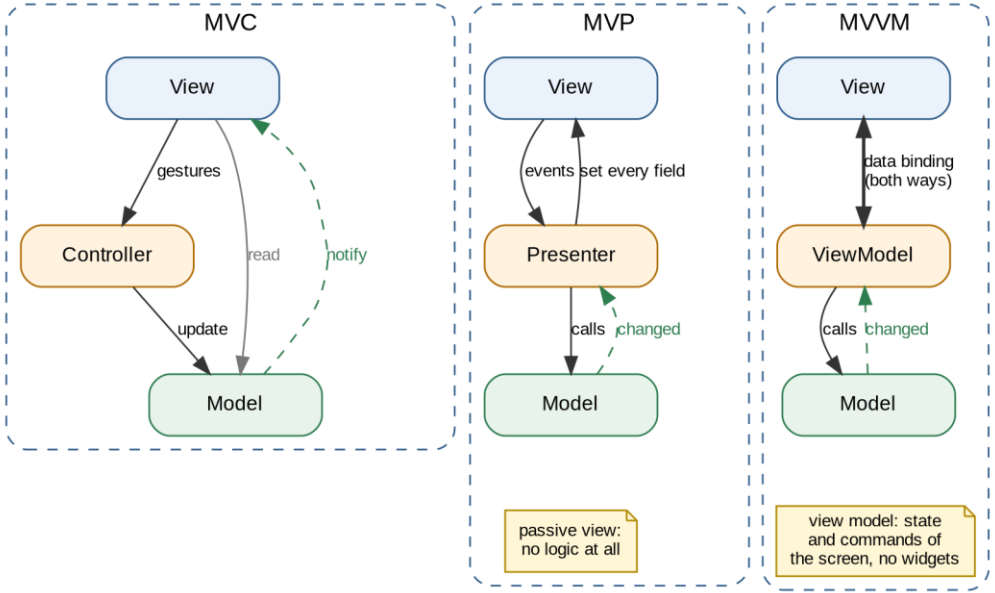


Figure 11.14 MVC and two of its descendants. In MVC the view observes the model and sends gestures to the controller. In MVP a passive view reports events to a presenter, which sets every field of the view. In MVVM the view is bound, in both directions, to a view model that holds the screen's state and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

Figure 11.14: in **MVP** a passive view reports to a presenter that sets every field — our laundry desk is strictly MVP. In **MVVM** the view is *bound* to a view model of observable properties.

Book: Section 11.11 and Figure 11.14.

Profitina in Practice

Profitina's web application (Python, FastAPI) shows MVC at scale. The **dashboard** is many views over different models — posts, feedback, the PROFIT ledger, subscriptions — assembled by one `GET /api/me/dashboard`.

That is Equation (11.1): a dashboard that stored its own copy of the counts would one day disagree with the channel it summarizes.

The community feed uses **Server-Sent Events** — a subject notifying listeners across a network; posting is optimistic, undone on failure.

Well over a thousand automated tests, mostly of models and handlers without a browser, gate every release. The platform advises; the user decides.

Book: Section 11.12.

Common Mistakes

Business rules in a button listener; a model that imports the GUI toolkit; forgetting to draw the first picture.

Notifying before validating, or once per field; changing the listener list while notifying (iterate over a copy).

Subscribing a lambda you cannot unsubscribe (the *lapsed listener*); creating Swing components off the event dispatch thread; the loop-variable lambda (`lambda k=key:`).

Believing the self-test proves the window is right; a class called `View` doing the controller's work. Ask of every method: does it draw, decide, or translate?

Book: Section 11.13.

Summary: Five Things to Remember

1. Model = data and rules, no GUI imports; views = presentation, no rules; controller = translation of gestures into model operations.

2. A view is a function of the model, $V = R(S)$; several views of one model can never disagree.

3. Observer: the model notifies without knowing its views — `PropertyChangeSupport` in Java, a list of callables in Python; notify once per logical change, over a copy of the list.

4. Model methods: validate, change, notify, return. The controller turns text into values; the model judges the values.

5. View contracts make a `--selftest` and fast unit tests possible; MVP and MVVM keep the rules out of the window too.

Book: Section 11.14.

Next Time

Chapter 12 — Evaluation and Discussion III: Predictive Text with Lists, Maps, Trees, and a Keypad. No new theory: Chapters 9, 10 and 11 worked through problems, step by step.

The keypad of today, rebuilt on four dictionaries — a slow prototype, a sorted list, a map and a tree — measured against each other, then put in MVC.

Before the meeting: Exercises 11.1, 11.3, 11.4 and 11.8, and read Sections 12.1 to 12.3.

Book: Section 11.15 and Chapter 12.

Chapter 1

Lecture 12 — Evaluation and Discussion III: Predictive Text with Lists, Maps, Trees, and a Keypad

Object-Oriented Programming (EF234302)

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*

Lecturer: Irfan Subakti — yifana@gmail.com

Book: Chapter 12, Evaluation and Discussion III: Predictive Text with Lists, Maps, Trees, and a Keypad. Every page of this handout ends with a pointer into the book, which is the primary reference.

Today's Meeting

1. No new language features. Chapters 9, 10 and 11 consolidated by working sixteen problems, each from a blank page, in Java and in Python (Section 12.1).

2. A method for collections-and-interface problems, in six steps (Section 12.2).

3. One project, worked end to end: the predictive text of old phones — four dictionaries, their timings, and a keypad in MVC (Sections 12.3–12.7, Problems 12.1–12.10).

4. Six new problems set in Profitina Laundry: a collection chosen by measuring, a bounded generic class, a comparator chain, a tested counter, two observer bugs, buttons made in a loop (Sections 12.8–12.9, Problems 12.11–12.16).

5. The problems compared, Profitina in practice, the common mistakes and the summary (Sections 12.10–12.13).

Book: Chapter 12, Sections 12.1 to 12.13.

Where We Stand

Chapter 9: collections — lists, sets, maps — and generics; `Comparable`, `Comparator` and `key=`.

Chapter 10: windows and events; listeners and callbacks, often lambdas or inner classes.

Chapter 11: Model–View–Controller; the model tells every registered view when it changes — the observer pattern.

Every problem follows one route: **the problem**, **in our own words**, **design**, **the Java solution**, **the Python solution**, **line by line**, **a tempting wrong solution**, **check yourself**.

> **HOW TO USE THIS MEETING.** Read a problem, cover the rest, spend ten minutes. For a dictionary problem, first write down *which operations* and *how often*; for a GUI problem, *what the model must remember*. Then compare.

Book: Section 12.1.

A Method in 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. $O(1)$ does not depend on n; $O(\log n)$ grows by one step when n doubles; $O(n)$ doubles when n doubles.

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?

Table 12.1: "store the dictionary" is not a specification; "find the words of a signature at every key press" is.

Book: Section 12.2 and Table 12.1.

Keys, Letters and Signatures



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.

Figure 12.1: eight keys carry letters; 7 and 9 carry four. With $\kappa(c)$ the key of letter c, a word's signature is Equation (12.1):

$$\text{sig}(c_1 c_2 \ldots c_m) = \kappa(c_1)\,\kappa(c_2)\,\ldots\,\kappa(c_m) \tag{12.1}$$

Book: Section 12.3.1, Figure 12.1 and Equation (12.1).

The Dictionary and the Design

sig(hello) = 43556; sig(home) = sig(good) = sig(gone) = 4663 — so the phone offers candidates: `*` next, `0` accept, `#` delete.

`words.txt` is deliberately dirty: one rule for every program — trim, lower-case, keep only non-empty a–z words. 93 of 99 lines survive.

Java packages `predictive` and `predictivegui`: `javac predictive/*.java ...`, then `java predictive.Sigs2WordsList 4663 43556`. `PREDICTIVE_DICT` selects the dictionary file.

Table 12.2, next page, maps the ten project problems to four parts.

Book: Sections 12.3.1–12.3.3.

The Whole Design at a Glance

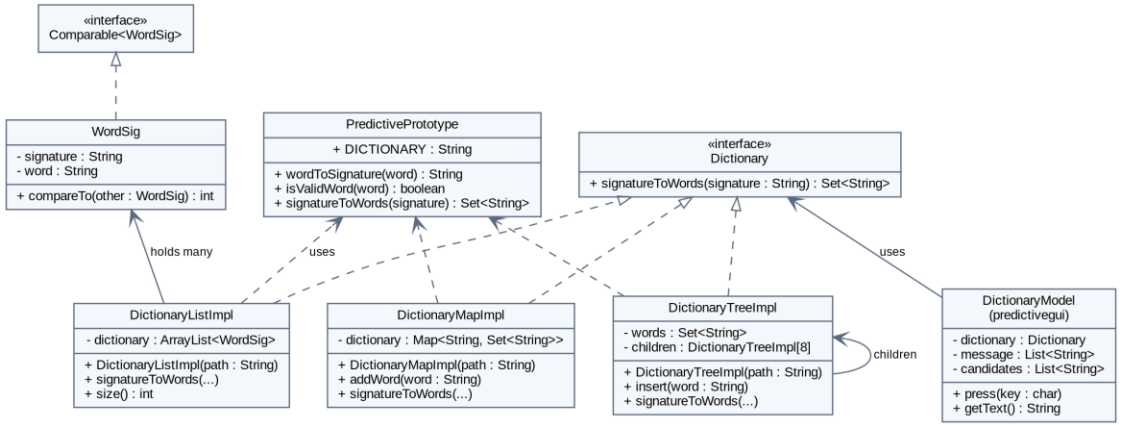


Figure 12.2 UML class diagram of the predictive-text project. Dashed lines with hollow triangles mean "implements"; solid arrows mean "has" or "uses"; `DictionaryTreeImpl` refers to objects of its own class.

Figure 12.2: one interface, `Dictionary`, three implementations — a sorted list, a hash map, a tree.

Book: Section 12.3.4 and Figure 12.2.

Four Parts, Ten Problems

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

Book: Section 12.3.4 and Table 12.2.

Problem 12.1 — From a Word to Its Signature

> **THE PROBLEM.** Write `public static String wordToSignature(String word)`: `"home"` gives `"4663"`; a character that is not a letter becomes a space. Build the result in a `StringBuffer`, and explain why.

Walk the word left to right; for each character, append its key. **Table 12.3** traces `"Home"`:

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`

Book: Section 12.4.1 and Table 12.3.

Why a StringBuffer?

A `String` is immutable: `s = s + '6'` copies every character so far. A word of n letters costs Equation (12.2), about $n^2/2$ copies:

$$1 + 2 + \cdots + n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} \tag{12.2}$$

A `StringBuffer` grows in place: the total stays proportional to n (**Listing 12.1**).

Listing 12.1 `PredictivePrototype.java` (excerpt): `wordToSignature` with a `StringBuffer`, and `keyOf` as a switch expression.

```
    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';
            default -> ' ';
        };
    }
```

Book: Section 12.4.1 and Listing 12.1.

A Tempting Wrong Solution: Arithmetic

"Three letters per key": $\kappa_{\text{wrong}}(c) = 2 + \lfloor \text{index}(c)/3 \rfloor$. For `"you"` it prints `:68`:

```
wrong 'you'      : :68   right: 968
wrong 4663 list  : ['gone', 'good', 'goof', 'home', 'Home', 'hone', 'hood', 'hoof']
```

Key 7 carries *four* letters, so from `s` on the formula runs one key ahead — wrong for exactly s, v, y, z. It passes every test built from "home" and "hello". A table cannot make this mistake.

Python: `str` is immutable too; collect digits in a list and `"".join(...)` once. CPython's in-place `+=` shortcut vanishes with one extra reference.

> **CHECK YOURSELF.** `wordToSignature("Laundry")`? `"e-mail"`? — `"5286379"` and `"3 6245"` (the hyphen keeps its position as a space).

Book: Section 12.4.1 and Listings 12.2 and 12.4.

Problem 12.2 — From a Signature to Its Words, Slowly

> **THE PROBLEM.** Write `Set<String> signatureToWords(String signature)` that reads the dictionary file on every call, returns lower-case words without duplicates, and explain why it is inefficient.

A `Set` enforces "no duplicates"; try-with-resources closes the file whatever happens. One lookup reads everything, Equation (12.3):

$$T_{\text{proto}}(n) = O(n\,\bar{m}) \quad \text{per lookup,} \tag{12.3}$$

4663: gone, good, goof, home, hone, hood, hoof; 5477: kiss and lips (**Listing 12.2**).

Listing 12.2 `PredictivePrototype.java` (excerpt): the slow lookup that reads the whole file for every question.

```
    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;
    }
```

Book: Section 12.4.2 and Listing 12.1.

Problem 12.2: the Traps

Wrong: keep matching lines in a `List` without lowering or trimming — the answer holds `Home` next to `home`, and `soap` is lost. Clean test data finds neither fault.

Wrong: closing by hand, `finally { fileScan.close(); }` — a missing file leaves `fileScan` null, and the `NullPointerException` hides the real error.

`&&` stops at the first `false`, so invalid lines are never converted; strings are compared with `equals`, never `==`.

> **CHECK YOURSELF.** Add one cheap test that skips most words. Does Equation (12.3) change? — Test `word.length() == signature.length()` first; the whole file is still read, so the cost stays proportional to n.

Book: Section 12.4.2.

Problem 12.3 — Two Command-Line Programs

`Words2SigProto home hello world my name is` and `Sigs2WordsProto` with the signatures it prints (**Listing 12.3**); output sorted (a `TreeSet`, `sorted()`) because hash order is not reproducible:

Listing 12.3 `Sigs2WordsProto.java`: the words of every signature on the command line, one line per signature, in alphabetical order.

```java
// 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)));
        }
    }
}
```

```
4663: [gone, good, goof, home, hone, hood, hoof]
43556: [hello]
96753: [world]
69: [my, ow, ox]
6263: [mane, name, oboe]
47: [is]
```

Book: Section 12.4.3 and Listing 12.6.

Problem 12.3: Printing the Arguments

Wrong in Java: `"input: " + args` prints `input: [Ljava.lang.String;@659e0bfd` — arrays do not override `toString()`. Use `Arrays.toString(args)`.

Wrong in Python: `print("input:", sys.argv)` starts with the program's file name. `sys.argv[0]` is the program; Java's `args[0]` is already the first argument.

> **CHECK YOURSELF.** `Words2SigProto don't panic`? — `input: [don't, panic]` and `output: 366 8 72642`; pasted into `Sigs2WordsProto`, the space in `366 8` makes three arguments.

Book: Section 12.4.3.

Problem 12.4 — A Sorted List and Binary Search

> **THE PROBLEM.** `DictionaryListImpl` reads the file once into an `ArrayList<WordSig>`, sorted once in the *numerical* order of the signatures; `signatureToWords` uses `Collections.binarySearch` and returns *all* matches, scanning above and below the index found.

Cards sorted by signature, like a telephone book: cards with the same signature lie together.

Trap: `String.compareTo` puts `"22"` before `"3"`. Signatures have no leading zeros, so fewer digits means smaller — Equation (12.4):

$$a < b \Leftrightarrow |a| < |b| \text{ or } (|a| = |b| \text{ and } a <_{\text{str}} b) \tag{12.4}$$

Book: Section 12.5.1 and Equation (12.4).

The Order in Code

Listing 12.4 `WordSig.java` (excerpt): `compareTo` in the numerical order of the signatures, without converting to a number.

```
    @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);
    }
```

Listing 12.4: lengths are compared with `Integer.compare`, which cannot overflow; equal lengths fall back to the string order. Both fields are `private final`: changing a card inside a sorted list would break the order binary search relies on.

Python: `__lt__` compares the tuples `(len(signature), signature)` — the same rule in one line.

Book: Section 12.5.1 and Listing 12.9.

Worked Example: Binary Search for 4663

Every look halves the part still to be searched, so at most Equation (12.5) cards are examined — 7 for 93 cards, 18 for 200,000:

$$\lceil \log_2(n+1) \rceil \tag{12.5}$$

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

Table 12.4: the search stops at `goof`, index 40, in the *middle* of the block 38–45. Walking down finds `good`, `gone`; walking up `home` (twice), `hone`, `hood`, `hoof`.

Book: Section 12.5.1, Equation (12.5) and Table 12.4.

Binary Search Plus a Walk

Listing 12.5 `DictionaryListImpl.java` (excerpt): one binary search, then a walk down and a walk up.

```
    @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());
        }
        for (int i = index + 1; i < dictionary.size() && matches(i, signature); i++) {
            result.add(dictionary.get(i).getWord());
        }
        return result;
    }
```

Listing 12.5: the probe `new WordSig(signature, "")` needs no word; not found gives $-(\text{insertion point}) - 1$. The costs are Equation (12.6):

$$T_{\text{build}}(n) = O(n \log n), \qquad T_{\text{list}}(n) = O(\bar{m} \log n + k) \quad \text{per lookup}, \tag{12.6}$$

Book: Section 12.5.1 and Listing 12.10.

Problem 12.4: Two Tempting Wrong Solutions

```
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 2222222222222223:aaaaaaaaaaaaaaad
```

Stop where binary search stops: `[goof]` — one word of seven. Always test a signature with several words.

Compare as numbers: a `double` holds 15–16 digits; `internationalization` has 20 letters. Three 17-digit signatures compare *equal*, and the search lands on `...223` when asked for `...222`. Equation (12.4) is exact.

> **CHECK YOURSELF.** For 5477 (kiss, lips), what does the first look decide? — `4766:iron` is smaller, so `low = 47`; wherever the search lands in the two-card block, the two-sided walk finds both.

Book: Section 12.5.1 and Listing 12.11.

Problem 12.5 — The Dictionary Interface

For every public member of the list, ask: does a *user* need it? Only the question “which words match?” — plus a precise contract for the awkward cases (**Listing 12.6**).

Listing 12.6 `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);
}
```

Wrong: copying the first implementation's public face, `ArrayList<WordSig> getDictionary()` — the map has no such list. Design from the users' needs.

Book: Section 12.5.2 and Listing 12.15.

Problem 12.6 — The First Timing

`Sigs2WordsList` is the prototype program with one line changed. For the race: 200,000 invented words from a generator Equation (12.7), identical in Java and Python:

$$x_{i+1} = (1103515245\, x_i + 12345) \bmod 2^{31} \tag{12.7}$$

Building once, then 2,000 lookups (the prototype gets only 5):

```
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 prototype needs 62,283 µs per lookup, the list 2.70 — over twenty thousand times less. The list pays 420 ms once, the price of about seven prototype lookups.

Book: Section 12.5.3, Equation (12.7) and Listing 12.19.

The Race on the Command Line

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

Table 12.5: four seconds against one in Java, fifteen against one in Python — but the list's second is almost all start-up and building; its 53 lookups take about 150 µs.

Wrong: timing the six example signatures on `words.txt` — both report `real 0m0.16s`: that measures the JVM's start-up.

> **CHECK YOURSELF.** After how many lookups does the list pay off against the prototype? Against the map? — $420/62 \approx 7$; never against the map, which builds *and* answers faster. The list's advantage is order.

Book: Section 12.5.3 and Table 12.5.

Problem 12.7 — A Multi-Valued Map

> **THE PROBLEM.** `DictionaryMapImpl` stores a map from each signature to the collection of its words: fast lookup without scanning, and fast insertion. Choose the `Map` class and justify it.

A cabinet of drawers labeled with signatures. `HashMap`: hash code → bucket, $O(1)$ expected, no order. `TreeMap`: $O(\log n)$, sorted keys. We never need order: `HashMap`, with Equation (12.8):

$$T_{\text{map}}(n) = O(\bar{m} + k) \text{ expected per lookup}, \qquad T_{\text{insert}} = O(\bar{m}) \text{ expected}, \tag{12.8}$$

Book: Section 12.6.1 and Equation (12.8).

Find or Create the Drawer

Listing 12.7 `DictionaryMapImpl.java` (excerpt): `addWord` finds or creates the drawer; the lookup hands out a read-only view.

```
    /** 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 12.7: the class is `final` because its constructor calls the public `addWord`. Python: `setdefault(signature, set()).add(word)` and a `frozenset` copy.

Wrong: `map.put(signature, new HashSet<>(Set.of(word)))` *replaces* the drawer — only `home` survives for 4663. **Wrong:** returning the drawer itself — a caller's `clear()` empties the dictionary.

Book: Section 12.6.1 and Listing 12.23.

Problem 12.8 — A Tree That Matches Prefixes

> **THE PROBLEM.** `DictionaryTreeImpl`: up to eight branches per node, one per key 2–9; each node stores the words whose signature *begins* with its path; the root stores none. Lookups return matching words and prefixes, cut to the signature's length.

Roads: `good` is stored by driving 4, 6, 6, 3 and leaving a copy at *every* crossing — `4`, `46`, `466`, `4663`. So crossing `466` knows `good`, `gone` and `home`.

Such a tree, whose paths spell keys, is a **trie** (from re*trie*val).

Book: Section 12.6.2.

The Trie for Eight Words

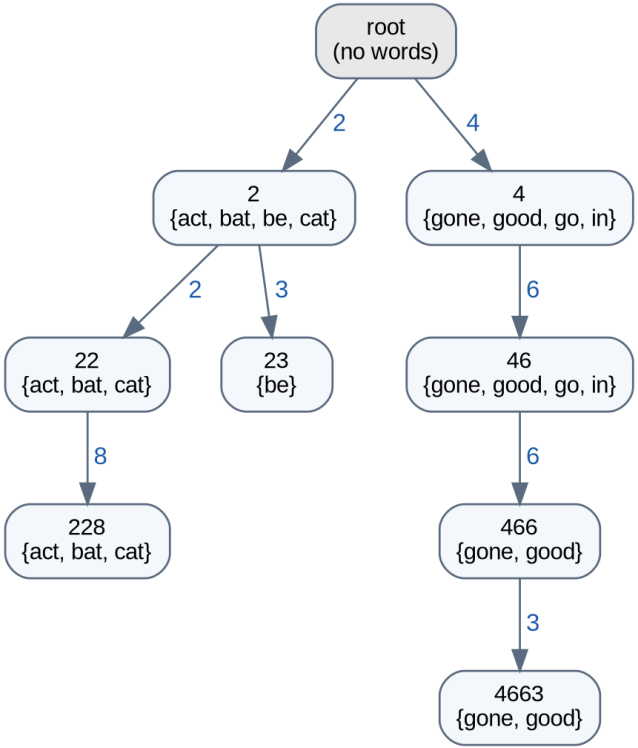


Figure 12.3 The tree dictionary for eight words. Each word is stored in every node on the path of its signature, so the node `46` holds all four words that begin with the keys 4 and 6.

Figure 12.3: `"228"` finds `{act, bat, cat}`; `"46"` finds four words, cut to `{go, in}`; `"229"` finds no road and returns an empty set.

Book: Section 12.6.2 and Figure 12.3.

Recursion by Delegation

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)`

Table 12.6 is Chapter 8's recipe. Each word is stored in $|w|$ nodes, giving Equation (12.9):

$$T_{\text{tree}}(s) = O(\ell + W_s\,\ell) \text{ per lookup}, \qquad \text{memory} = O\left(\sum_w |w|\right) \text{ word references}, \tag{12.9}$$

90 distinct words, 354 letters: 354 references in 120 nodes. The strings are shared, not copied.

Book: Section 12.6.2, Table 12.6 and Equation (12.9).

The Trie in Code

Listing 12.8 `DictionaryTreeImpl.java` (excerpt): `insert` adds the word to the *child*, then delegates; `find` follows the path.

```
    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);
    }
```

Listing 12.8: the word is added to the child, never to `this`, so the root stays empty. `trim` cuts every word to n letters in a set, so `good` and `goof` collapse into `goo`.

Book: Section 12.6.2 and Listing 12.27.

Problem 12.8: Prefixes, and the End-Only Tree

```
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 adds `worke` for 96753 (from `worked`, `worker`) and `ir` for 47 (from `iron`).

Wrong: storing each word only where its path *ends* — for 9675, halfway through `world`, it answers only `[work]`: the point of the tree is lost.

> **CHECK YOURSELF.** Insert only `home` and `hood`. Nodes? `signatureToWords("466")`? — 5 nodes (both are 4663), and `{hom, hoo}`.

Book: Section 12.6.2.

Problem 12.9 — One Line to Change

`Sigs2WordsList`, `Sigs2WordsMap` and `Sigs2WordsTree` differ only in the class after `new` (**Listing 12.9**):

Listing 12.9 `Sigs2WordsMap.java` (excerpt): the one line that differs is the one that creates the dictionary.

```
    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)));
        }
    }
```

Wrong: `DictionaryListImpl d = ...`, and later `d.size()` — now switching needs two changes, and the second does not compile for the map. Declare the interface type.

Book: Section 12.6.3 and Listing 12.31.

How Each Dictionary Scales

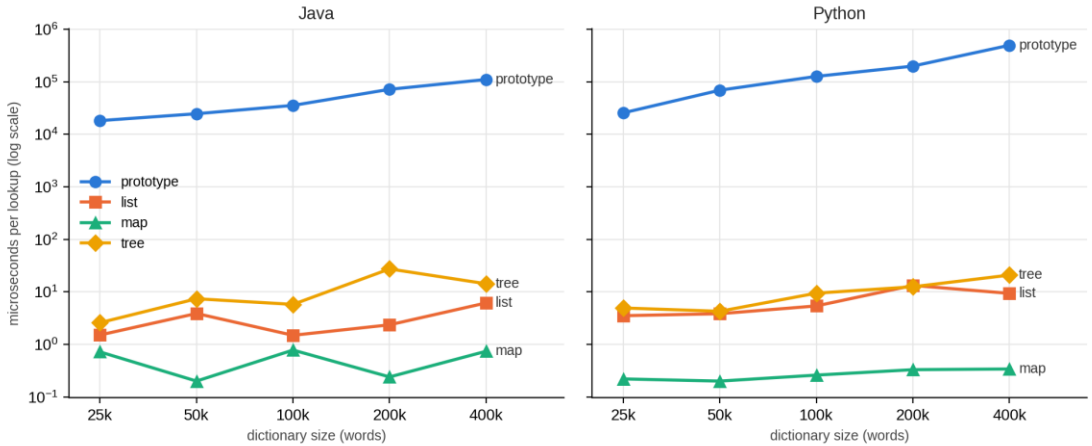


Figure 12.4 Microseconds per lookup against dictionary size, on logarithmic axes, measured with `TimingComparison` (left) and `timing_comparison.py` (right). The prototype rises with the size; the map stays flat; the list and the tree lie in between.

Figure 12.4: the prototype rises with the size, Equation (12.3); the map stays flat, Equation (12.8); the list creeps up with log$n$; the tree lies above the list because it answers a bigger question.

Book: Section 12.6.3 and Figure 12.4.

The Four Dictionaries Compared

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

Table 12.7: list versus map, and tree versus map, are noticeable with many lookups — but only the tree answers *which words the user could be typing right now*.

> **CHECK YOURSELF.** 13.57 µs per tree lookup against 0.22 for the map, at four key presses per second — does it matter? — No: 56 µs of work per second.

Book: Section 12.6.3 and Table 12.7.

Problem 12.10 — The Predictive-Text GUI

> **THE PROBLEM.** In package `predictivegui`, a GUI in MVC for any `Dictionary`. The model may store the dictionary and the words entered, but **no signatures or spaces**. The controller is twelve buttons; the view is a text window the user cannot type into. `2` gives a, b or c; `*` cycles; `0` completes the word; `#` deletes the last letter.

What must the model remember? The finished words, the candidates for the current word, and which candidate is on show. The digits pressed need not be stored: the word on show has one letter per key, so its signature *is* the digits.

Book: Section 12.7.1.

What Each Key Does to the Model

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

Table 12.8: a key that matches nothing is ignored — a phone that refuses an impossible key behaves like a phone. After 4, 6, 6 the tree gives `gon`, `goo`, `hom`, `hon`, `hoo`, `inn`.

Book: Section 12.7.1 and Table 12.8.

Model, View, Controller

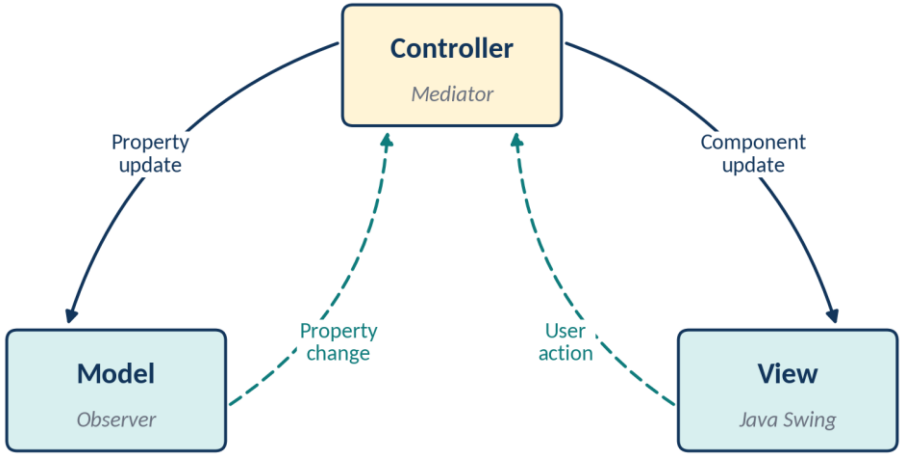


Figure 12.5 The Model–View–Controller pattern. The controller mediates: it turns user actions into updates of the model, and the model's changes reach the view.

Figure 12.5: the controller calls `model.press(key)`; the model changes and notifies its listeners; the view asks the model for the text. Notification uses a `List<Runnable>`, not the deprecated `Observable`.

Book: Section 12.7.1 and Figure 12.5.

A Label That Was Typed by Hand



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`.

Figure 12.6: key 9 reads `vwxy`, while **Figure 12.1** puts w, x, y, z on 9. The label was typed apart from the letters the signature code uses; the new controller takes its labels from one table.

Book: Section 12.7.1 and Figure 12.6.

The Model: the Signature Is Recomputed

Listing 12.10 `DictionaryModel.java` (excerpt): `typeKey` recomputes the signature from the word on show; `nextCandidate` wraps with `%`.

```
    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
        }
    }
```

Listing 12.10: "the digits so far, plus the new one" is `wordToSignature(currentWord()) + key`. `lookUp` sorts the answer through a `TreeSet`.

Book: Section 12.7.1 and Listing 12.35.

The Controller: Twelve Buttons from One Table

Listing 12.11 `KeypadController.java` (excerpt): one button per row of the table; each listener captures its own key.

```
    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);
        }
    }
```

Listing 12.11: `char key` is a *new* variable every turn, never reassigned, so the lambda may capture it (Problem 12.16). Python: `make_command(model, key)` gives each button its own function.

Book: Section 12.7.1 and Listing 12.37.

Worked Example: the Self-Test as a Film

```
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 "
```

`g`, `go`, `gon` (first of six prefixes), `gone`; `*` walks to `home`; `0` fixes it. Then `3` gives `worke` (from `worked`), and `#` takes the `e` back: 9675 looked up afresh gives `work`.

Book: Section 12.7.1 and Listing 12.38.

The Swing Window

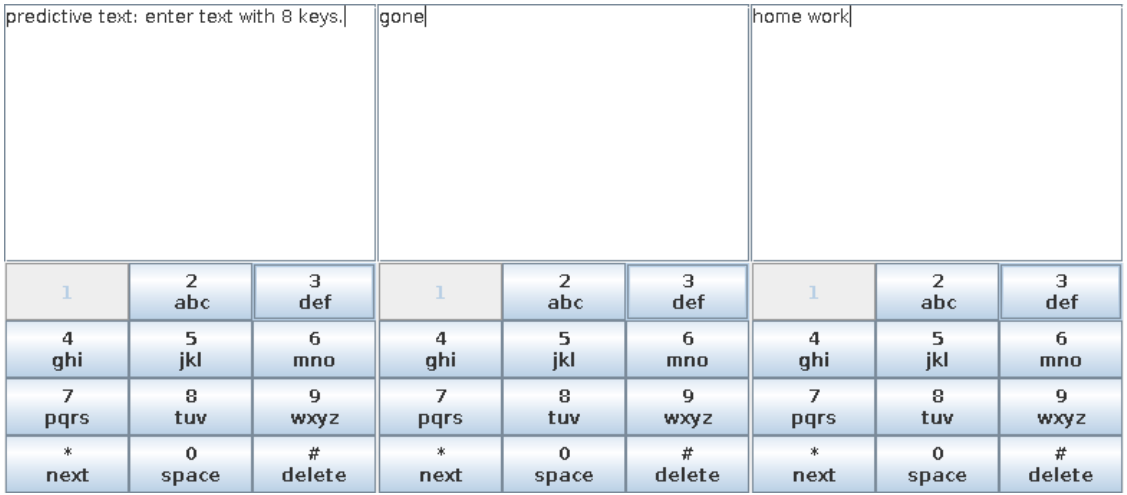


Figure 12.7 The Swing window at the start (left), after pressing 4, 6, 6, 3 (middle), and after `* * * 0 9 6 7 5 3 #` (right). Key 1 is disabled because it carries no letters.

Figure 12.7: the real window, opened on a virtual screen at three moments.

Book: Section 12.7.1 and Figure 12.7.

The Tkinter Window



Figure 12.8 The Tkinter window at the same three moments as in **Figure 12.7**.

Figure 12.8: the model imports nothing from Tkinter; the view and the controller import it inside their constructors, so the self-test runs where Tkinter is missing.

Book: Section 12.7.1 and Figure 12.8.

Problem 12.10: the Traps

Wrong: let the text area *be* the model — read it back with `getText()`, change it, write it. Then `*` guesses candidates from text, `#` guesses where the word begins, the user can corrupt the only copy, and nothing is testable without a window.

Wrong: store `currentSig` next to the words. Two fields describing one fact can disagree; one field recomputed on demand cannot.

The model is tested alone with a fake dictionary written as a lambda — `Dictionary` has one method (book Listing 12.39).

CHECK YOURSELF. After `4 3 5 5 6 0 # 9 6`? — `hello wo`: the `#` does nothing, because finished words are not edited.

Book: Section 12.7.1.

Problem 12.11 — Choosing a Collection by Measuring

THE PROBLEM. At the pickup counter, 100,000 ready order numbers: say at once whether a scanned number is ready; print all numbers in order at the end of the day; log each scanned number once, in first-seen order.

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

Table 12.9: one structure per job. Python: a sorted list with `bisect` replaces `TreeSet`; `list(dict.fromkeys(items))` keeps first-seen order.

Book: Section 12.8.1 and Table 12.9.

Problem 12.11: the Measurements

```
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
```

10,000 questions: `ArrayList` 1,470 ms, `HashSet` 5.5 ms — a factor of about 270 (Python: about 5,000). Only insertion-ordered and sorted collections *promise* an order.

Wrong: `if (!seen.contains(id)) seen.add(id);` on an `ArrayList` — $O(n^2)$, quadrupling with each doubling. One type change fixes it: `LinkedHashSet`.

Book: Section 12.8.1 and Listing 12.45.

Problem 12.12 — A Bounded Generic Class

`Bin<T extends Weighed>` holds items of one kind; `heaviest()` returns a `T`, so `shirts.heaviest().owner()` needs no cast (**Listing 12.12**).

Listing 12.12 `GenericBin.java` (excerpt): a generic class whose type parameter is bounded by `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>
        }
```

`totalKg(List<? extends Weighed>)` accepts a list of shirts or towels; `<T extends Comparable<? super T>> T maxOf(List<? extends T>)` is a generic method. Items are **records**.

Book: Section 12.8.2 and Listing 12.47.

Problem 12.12: Invariance, Again

Wrong: `static double totalKg(List<Weighed> things)`, called with a `List<Towel>`:

```
WrongTotal.java:12: error: incompatible types: List<Towel> cannot be converted to List<Weighed>
        System.out.println(totalKg(towels));
                                   ^
```

If a `List<Towel>` were a `List<Weighed>`, the method could add a shirt to the caller's towels. `? extends Weighed` gives up writing to gain flexibility; mypy says *"list" is invariant — Consider using "Sequence"*.

> **CHECK YOURSELF.** A method that adds three towels and accepts `List<Towel>`, `List<Weighed>` and `List<Object>`? — `static void addTowels(List<? super Towel> into)`: producer `extends`, consumer `super`.

Book: Section 12.8.2.

Problem 12.13 — A Chain of Comparators

Express first; then the earlier ready time; then the customer's name; then the order number. **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

Java: each key states its own direction (**Listing 12.13**); Python: the key `(not o.express, o.ready_at, o.customer, o.id)`.

Listing 12.13 `OrderSorting.java` (excerpt): the four-key board order; only the express key is reversed.

```
    /** 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);
```

Book: Section 12.8.3, Table 12.10 and Listing 12.49.

Problem 12.13: the Traps

```
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]
```

Wrong: every key forwards, then `.reversed()` at the end — the *whole* chain reverses: later ready times first, names Z to A. Reverse only the key that needs it.

Wrong: `(a, b) -> (int) (a.kg() - b.kg())` — 6.0 minus 5.2 is 0.8, cast to 0, "equal". Use `comparingDouble`.

CHECK YOURSELF. Heaviest first, equal weights by id ascending, in Python? — `key=lambda o: (-o.kg, o.id)`.

Book: Section 12.8.3.

Problem 12.14 — A Counter with a Tested Model

Bags handed in today; one machine takes 8 bags, so machines are bags divided by 8 *rounded up*, Equation (12.10):

$$\lceil \frac{b}{8} \rceil = \lfloor \frac{b+7}{8} \rfloor \tag{12.10}$$

Listing 12.14 `BagCounterApp.java` (excerpt): the model's rules — machines rounded up, never below zero, notify only on a real change.

```
        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 12.14: two label views read the *same* model, so they cannot disagree. **Wrong:** no model — parse the label, add one, write it back: after nine clicks, `Machines needed: 0`.

Book: Section 12.9.1, Equation (12.10) and Listing 12.51.

Problem 12.15 — Two Observer Bugs

Rule 1: change the state first, then notify. **Rule 2:** notify a *copy* of the listener list — a listener may unsubscribe itself (a one-shot alarm) (**Listing 12.15**).

Listing 12.15 `ObserverBugs.java` (excerpt): the careful branch and the buggy branch of `use`.

```
        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;
            }
        }
```

```
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
```

Book: Section 12.9.2 and Listing 12.53.

Problem 12.15: Loud Bug, Quiet Bug

Java's `ArrayList` detects the change and throws `ConcurrentModificationException`: the display misses the wash — *display shows 11 l, tank has 8 l*.

Python's `for` continues at the next index and *skips* the display without a word — *display shows 11 l, tank has 5 l*. The quiet version is the more dangerous one.

Wrong fix: catch the exception or loop with an index — the listener is still skipped. `CopyOnWriteArrayList` is made for listener lists.

> **CHECK YOURSELF.** A listener *adds* a listener during notification in the careful tank. Is the new one called now? — No: the loop walks the copy; it hears from the next event on.

Book: Section 12.9.2.

Problem 12.16 — Buttons Made in a Loop

Java captures values: only *effectively final* locals; `for (int i ...; i++)` with `keys.charAt(i)` in a lambda does not compile. **Python captures variables:** a lambda reads the value when it *runs* (**Listing 12.16**).

Listing 12.16 `keypad_closures.py` (excerpt): the late-binding bug, a default argument and a factory function.

```python
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]
```

Wrong in Java: a one-element array “fixes” the compile error — clicking 4, 6, 6, 3 records `####`, Python’s bug reintroduced by hand.

Book: Section 12.9.3 and Listing 12.56.

The Sixteen Problems Compared

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

Table 12.11 (Continued) *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
12.15	observer	change first, notify a copy	stale views; skipped listeners
12.16	closures	each listener takes its own key	late binding: every button presses #

Table 12.11: 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.

Book: Section 12.10 and Table 12.11.

Profitina in Practice

The right collection for the frequent question: “have I seen this page?” for every link — a hash set of canonical addresses; the weekly score trend — an ordered series exported to CSV.

One interface, many implementations: each *signal* is scored and explained behind one interface, like `Dictionary`; a new signal is a new class.

Models, views and observers: “N new” counters are views of a model; Server-Sent Events push changes; optimistic posting changes the local model first and restores it on failure.

The system suggests; the person chooses — as predictive text offers candidates and the user confirms one.

Book: Section 12.11.

Common Mistakes

Choosing a collection by habit; comparing numbers as strings or strings as numbers; keeping only the first binary-search hit.

Handing out a collection that belongs to an object; replacing instead of adding in a multi-valued map; expecting an order from a hash set.

A concrete type where the interface would do; reversing a whole comparator chain; subtracting to compare.

State in the view; notifying before changing, or walking a live list; capturing a changing loop variable; timing the wrong thing.

Book: Section 12.12.

Summary: Five Things to Remember

1. A collections problem is a problem about operations: list them with their frequencies, match each to a structure, hide the choice behind an interface.

2. Prototype $O(n\bar{m})$ per lookup; sorted list $O(\bar{m}\log n + k)$; hash map $O(\bar{m} + k)$ expected; trie — the only one that answers prefixes.

3. Measure the phase that differs, with large data, several times.

4. A GUI in MVC keeps each fact once, in a model tested without a window; views show, controllers forward.

5. Generics are invariant; comparator keys state their own directions; observers change first and notify a copy; listeners made in a loop capture their own values.

Book: Section 12.13.

Next Time

Chapter 13 — Iterators, Immutability, and More Collections. Today we walked collections with `for`; next we write the walker ourselves.

Iterators and `Iterable`, your own iterable collection, changing a collection while walking it; copies, immutability from zero, value objects; streams, comprehensions and `itertools`.

Before the meeting: Exercises 12.1, 12.3, 12.6 and 12.8, and read Sections 13.1 to 13.3.

Book: Section 12.14 and Chapter 13.

Lecture 13 — Iterators, Immutability, and More Collections

Object-Oriented Programming (EF234302)

Chapter 13 of the book, in miniature, for the classroom.

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*.

Lecturer: Irfan Subakti — yifana@gmail.com

Today we lift the lid on the `for` loop, learn to sort objects that have no obvious order, and meet objects that cannot change, which turn out to be one of the most useful tools we have. Book: Chapter 13, opening paragraph.

Agenda

1. **Walking.** The iterator protocol in Java and Python, the for-each loop, `ListIterator`, and iterables of our own, written by hand and as Python *generators*.
2. **Changing while walking.** Why Java throws `ConcurrentModificationException`, why Python quietly skips elements, and the safe patterns.
3. **Order.** `Comparable`, `Comparator`, Python key functions, the contract every comparison must keep, and a comparison that only looks right.
4. **Copies and immutability.** Aliasing, shallow and deep copies; immutable classes, records, frozen dataclasses, views and defensive copies.
5. **Values and pipelines.** Value objects and hashing; streams, comprehensions and `itertools`; the Profitina Laundry ledger.

The pages follow the book in order, and the exercises at the end of the chapter are the homework. Book: Sections 13.2 to 13.11.

Where We Left Off

Chapter 9 gave us the three great families of collections, lists, sets and maps, and the generics that let a `List<String>` hold only strings.

Chapters 10 and 11 put windows on the screen and split a program into model, view and controller; **Chapter 12** worked problems that combined all of it.

Throughout, we used collections the simple way: add, look up, and walk with a `for` loop, **without asking how the loop visits each element**.

> ✔️ **KEY INSIGHT.** An *iterator* separates **walking** a collection from **storing** it; *immutability* separates **creating** an object from **using** it. Each piece of the program has fewer things to know and fewer ways to go wrong.

The laundry now wants a history of every order, one customer's orders, a weekly report, and prices that can never change by accident. Book: Section 13.1.

What Is an 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.

A `HashSet` has no "element number 2"; a tree or a file read line by line has no index at all. We need one way that works for **every** collection: *give me the elements one at a time until there are none left.*

Java: `java.util.Iterator<E>` with `boolean hasNext()` and `E next()`; `next()` past the end throws `NoSuchElementException`. `remove()` is optional. Every collection has `iterator()`.

The *Iterator pattern* of Gamma et al. goes back to the language CLU of the 1970s; both Java and Python build their `for` loops on it.

Book: Sections 13.2 and 13.2.1.

The Protocol in Java

Listing 13.1 `IteratorBasics.java`, block 1: printing every string with an explicit iterator.

```
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());
        }
```

```
Luffy, Monkey D.
Zoro, Roronoa
Robin, Nico
```

In Listing 13.1, `crew` is declared as the most general type that does the job, `Collection`. `iterator()` creates a separate object; `hasNext()` is asked before every step; `next()` returns the string *and* moves on.

Calling `next()` twice in one loop body consumes two elements. Book: Section 13.2.1 and Listing 13.1.

One Element Too Many

Listing 13.2 `IteratorBasics.java`, block 4: a fourth `next()` on a three-element collection.

```
        // 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());
        }
```

```
hasNext() after three calls to next(): false
fourth next(): NoSuchElementException
```

Block 2 of the same program changes a local copy: `element += " and Zoro, Roronoa"` builds a new string and rebinds `element`; the collection still prints `Luffy, Monkey D.`. Block 3 sums `[3, 2, 4]` to 9 with a for-each loop.

In Listing 13.2, a correct program never meets `NoSuchElementException`: it asks `hasNext()` first. With mutable elements, `element.setName(...)` would reach the object inside the collection (Section 13.7). Book: Section 13.2.1.

The Cursor Sits Between Elements

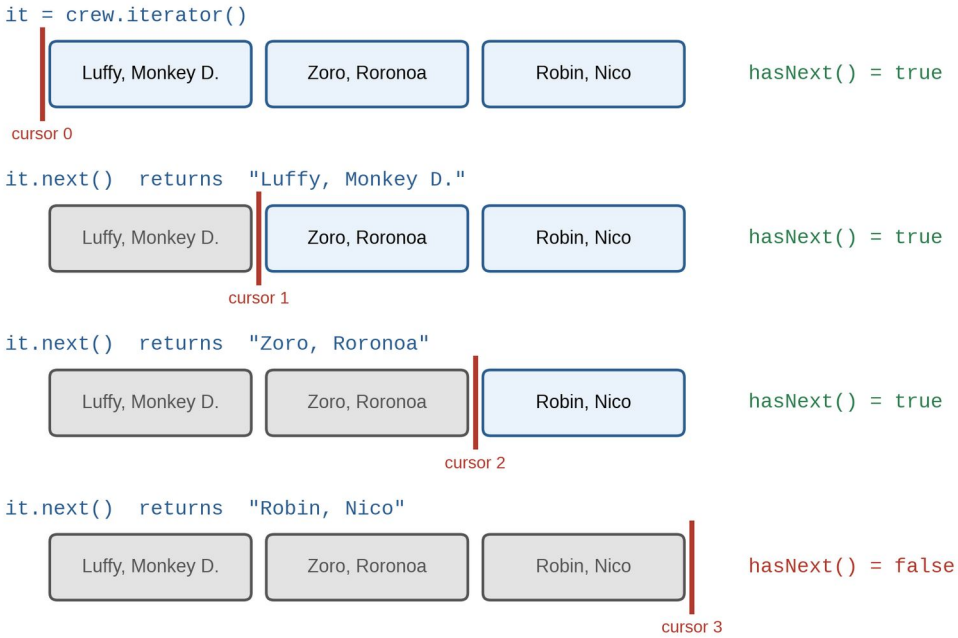


Figure 13.1 An iterator's cursor sits between elements. Each `next()` jumps over one element and returns it (grey boxes are already returned); after the last jump, `hasNext()` is `false`.

In Figure 13.1, number the gaps $0, 1, \ldots, n$. The cursor c starts at 0 and each `next()` adds one:

$$\text{hasNext}() \Leftrightarrow c \neq n \tag{13.1}$$

Equation (13.1) looks too obvious to write down; in a few pages it explains a bug that silently skips an element. Book: Section 13.2.1, Figure 13.1 and Equation (13.1).

The Same Protocol in Python

Listing 13.3 `iterator_basics.py`, first half: `iter()`, `next()` and `StopIteration`, written out the way a `for` loop does it behind the scenes.

```python
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)
```

Listing 13.3: there is no `hasNext()`; you *try* to take the next element, and `StopIteration` is the answer "there is none" (PEP 234). `next(it, "no more")` returns a default instead; `iter(it) is it` is `True`.

Nobody writes the `while True` loop in real code, but writing it once shows that `for element in crew:` is no magic. Book: Section 13.2.2.

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`

Table 13.1: a plain iterator only moves **forward** and only **reads**. That modesty is a feature: every collection can keep so small a promise, even a sequence computed on demand and never stored.

Book: Sections 13.2.2 and 13.2.3, and Table 13.1.

For-Each Is an Iterator in Disguise

The Java Language Specification defines the enhanced loop by translation. `for (Integer i : c) { sum += i; }` becomes:

```
for (Iterator<Integer> $it = c.iterator(); $it.hasNext(); ) {
    Integer i = $it.next();
    sum += i;
}
```

Anything that implements `Iterable<T>` (one method, `Iterator<T> iterator()`) may appear after the colon. Python's `for i in c:` is translated the same way into the loop of Listing 13.3.

> ⚠ **WARNING.** Inside a for-each loop you have **no handle on the iterator**, so you cannot call its `remove()`.

For an array, Java's enhanced loop uses an index instead, because arrays do not implement `Iterable`. Book: Section 13.3.

Removing Adjacent Duplicates

Listing 13.4 `RemoveDuplicates.java`, the classic method: remember the previous element, collect the unique ones in a new list.

```java
// 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;
}
```

Two quiet defects in Listing 13.4, from one careless assumption. **Empty list:** `input.get(0)` throws `IndexOutOfBoundsException`. **`null` as a marker:** for `[null, null, "Nami"]`, `previous` stays `null` for ever, and `"Nami"` is lost.

Whenever a real value such as `null`, `0` or `-1` is made to mean "nothing yet", ask whether it can also occur as data; if it can, use a separate flag. Book: Section 13.3.1.

In Place, with ListIterator

Listing 13.5 `RemoveDuplicates.java`, the in-place method: `ListIterator.remove()` deletes the element `next()` just returned.

```
    // 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;
        }
    }
```

The safe copying version, `dedupSorted`, fixes both defects with a separate `boolean first` and `Objects.equals`. In Listing 13.5, `ListIterator` also moves backwards (`hasPrevious`, `previous`), reports indexes, and can `set`, `add` and `remove` at the cursor; `listIterator(list.size())` starts after the last element. Book: Sections 13.3.1 and 13.3.2.

Duplicates in Python

Listing 13.6 `remove_duplicates.py`: the flag version, and deletion in place walking the indexes from the end.

```
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]
```

In Listing 13.6, deleting `items[i]` shifts only elements *after* `i`, which the loop has already visited. `[key for key, _ in groupby(my_list)]` gives the same answer in one line; `lst[i] = e` does the job of `ListIterator.set()`.

Python has no `ListIterator` because it does not need one: a list is indexed, and `reversed(range(len(xs)))` walks backwards. Book: Section 13.3.2 and Listing 13.4.

An Iterable of Our Own

12.5 kg of washing, machines of 5 kg: the plan is 5, 5, 2.5. The plan never needs to **store** its loads; the next one is computed from what is left.

Listing 13.7 `LoadPlanDemo.java`, the collection `LoadPlan`: it holds only what defines the plan, and `iterator()` returns a new `LoadIterator`.

```
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();
    }
```

*In Listing 13.7, keep the **collection** (what defines the data) apart from the **iterator** (the position of one walk). Book: Sections 13.4 and 13.4.1.*

The Inner Iterator Class

Listing 13.8 `LoadPlanDemo.java`, the private inner class `LoadIterator`, which reads the fields of its `LoadPlan` directly.

```
    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;
        }
    }
```

```
a is at its third load: 2.5 kg
b is still at its first: 5.0 kg
```

*Listing 13.8 shows three properties in the output: the for-each loop works, iterators are **independent** (`a` and `b`), and the plan is **lazy**: 133,334 loads of a million-kilogram plan are computed one by one and never stored. Book: Section 13.4.1.*

Python: A Generator Writes the Iterator

Listing 13.9 `load_plan.py`, `GeneratorLoadPlan`: the same iterable, with `__iter__` written as a generator function.

```
class GeneratorLoadPlan(LoadPlan):
    """The same iterable; a generator function writes the iterator for us."""

    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()
```

The same file also writes `LoadIterator` by hand, with `__init__`, `__iter__`, `__next__` and an explicit `StopIteration`; in Listing 13.9 it has become a four-line loop with `yield` (PEP 255). Book: Section 13.4.2.

Watching a Generator Pause

DEFINITION. A **generator function** is a Python function whose body contains `yield`. Calling it returns a **generator**, an iterator that runs the body step by step: each `next()` resumes the body, runs it to the next `yield`, returns the value and suspends again; when the body ends, the generator raises `StopIteration`.

```
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]
```

After `g = iter(lazy)` nothing has been computed; `next(g)` runs to the first `yield`; `list(g)` drives the generator to its end.

Java has no `yield` for iterators; the explicit inner class of Listing 13.8 remains its standard way. Book: Section 13.4.2.

An Iterable Is Not an Iterator

`it = iter(plan)`: `list(it)` gives `[5.0, 5.0, 2.5]`, and a second `list(it)` gives `[]`. The iterator is **used up**; there is no rewind.

`list(plan)` works every time: `plan` is an **iterable**, and each call to `plan.__iter__()` returns a fresh iterator.

Java's type system enforces the difference: for-each accepts an `Iterable`, never an `Iterator`. Python's `for` accepts both, so a function that loops twice over a generator **silently does nothing the second time**.

> ✔️ **KEY INSIGHT.** To walk something more than once, hold an **iterable** (a list, a tuple, an object like `LoadPlan`); if in doubt, call `list(...)` once and keep the list.

`itertools.islice(huge, 2)` takes two elements lazily, and `sum(1 for _ in huge)` counts without building a list. Book: Section 13.4.3.

Changing a Collection While Walking It

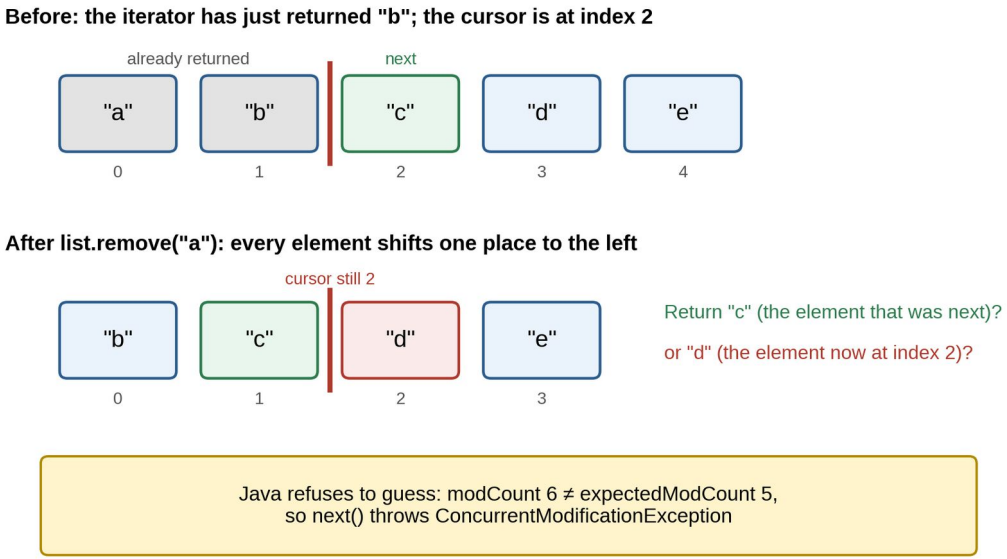


Figure 13.2 Why Java throws `ConcurrentModificationException`. After `remove("a")` the elements shift left, but the iterator's cursor does not; there is no right answer to "what comes next?", so the iterator refuses to answer.

In Figure 13.2 the iterator has returned `"b"`; someone calls `list.remove("a")`. Should `next()` return `"c"` or `"d"`? There is no good answer, so Java's collections **refuse to guess**.

"Concurrent" here does not mean two threads (that is Chapter 14): it means the change happened during the iteration. Book: Section 13.5 and Figure 13.2.

How the Iterator Notices

An `ArrayList` counts every **structural** change (`add`, `remove`, but not `set`) in `modCount`. A new iterator copies it into `expectedModCount`; its own `remove()` updates both.

At the start of every `next()` it checks

$$\text{modCount} = \text{expectedModCount} \tag{13.2}$$

and throws `ConcurrentModificationException` if Equation (13.2) is false.

So while a for-each loop walks a `List`: `set` is **allowed**, `add` and `remove` through the list are **forbidden**.

Book: Section 13.5 and Equation (13.2).

A Silent Skip

Listing 13.10 `ConcurrentModification.java`, block 4: removing the second-to-last element ends the loop early, with no exception.

```
        // 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());
```

```
no exception, but visited only: Ace Luffy Sanji Zoro
```

In Listing 13.10: after `next()` returned `Zoro`, the cursor $c=4$; the removal shrinks the list to $n=4$. By Equation (13.1), `hasNext()` asks $4 \neq 4$: false. The loop **ends** before `next()` can check Equation (13.2), and Nami is never visited.

Blocks 1–3 of the same program prove the rules: `set` is fine, `add` and `remove` throw on the next hidden `next()`. The check is only "best effort". Book: Section 13.5 and Listing 13.7.

The Right Ways

Listing 13.11 `ConcurrentModification.java`, block 5: removal through the iterator itself, and `removeIf`.

```
        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);
```

```
Iterator.remove(): [Luffy, Sanji, Zoro, Nami]
removeIf():        [Luffy, Sanji, Zoro, Nami]
```

Listing 13.11: the iterator's own `remove()` updates `expectedModCount`, so everything stays consistent; `removeIf(predicate)` says what we mean in one line.

To **add** during a walk, iterate over a copy: `for (String name : new ArrayList<>(myList))`.

*The rule is not "catch `ConcurrentModificationException`" but **never change a collection's size while a for-each loop walks it**. Book: Section 13.5.*

Python Does Not Check

Listing 13.12 `concurrent_modification.py`, blocks 2 and 3: removing from a list during a `for` loop gives wrong answers, and no error.

```
    # 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)
```

EXAMPLE. In Listing 13.12, remove the evens from `[2, 4, 6, 7]`. Index 0: remove 2, the list is `[4, 6, 7]`. Index 1 now holds **6**, so **4 is skipped**. Remove 6: `[4, 7]`. Index 2 is past the end. Result `[4, 7]`, not `[7]`.

Dictionaries and sets are stricter: changing their size while iterating raises `RuntimeError: dictionary changed size during iteration`*. Book: Section 13.5 and Listing 13.8.*

Safe Ways to Change While Walking

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`

In Table 13.2, `lst[:] = [...]` replaces the contents **in place**, so everyone else holding the list sees the change; `list(stock)` is a copy of the keys taken before the loop starts.

Book: Section 13.5 and Table 13.2.

Comparable and Comparator

`Comparable<T>`: the class has one **natural order** that everyone agrees on (numbers, strings, dates), via `int compareTo(T other)`.

`Comparator<T>`: a *separate* object with `int compare(T a, T b)`. Students by name for a register, by age for a team, by marks for a prize: one natural order, any number of comparators.

Only the **sign** of the result matters: negative "first comes before", zero "equal in this order", positive "first comes after". With sgn the sign:

$$\operatorname{sgn}(\operatorname{cmp}(a,b)) = -\operatorname{sgn}(\operatorname{cmp}(b,a)) \text{ for all } a,b \tag{13.3}$$

$$\operatorname{cmp}(a,b) > 0 \text{ and } \operatorname{cmp}(b,c) > 0 \Rightarrow \operatorname{cmp}(a,c) > 0 \tag{13.4}$$

Equation (13.3) is antisymmetry, Equation (13.4) transitivity; the third rule says that if $\operatorname{cmp}(a,b) = 0$ *then* a *and* b *compare the same way with everything else. Book: Sections 13.6 and 13.6.1.*

Never Compare by Subtraction

`return s1.getAge() - s2.getAge();` has the right sign for ages. But an `int` holds only -2^{31} to $2^{31}-1$:

$$2^{31}-1-(-1)=2^{31} \overset{\text{overflow}}{\rightarrow} -2^{31}<0 \tag{13.5}$$

By Equation (13.5) the largest `int` is declared *smaller* than -1, which breaks Equation (13.3).

> ✔ **KEY INSIGHT.** The cure costs nothing: `Integer.compare(a, b)`, `Long.compare`, `Double.compare`, or `Comparator.comparingInt`.

Book: Section 13.6.1.

A Comparator by Age

Listing 13.13 `AgeSort.java`, the hand-written `AgeComparator`: `null` first, then `Integer.compare`.

```
    // 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 13.14 `AgeSort.java`, the same orders built from parts: `comparingInt`, `reversed`, `nullsFirst`.

```
        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);
```

Listing 13.13 treats the special cases first; Listing 13.14 shows why the combinators exist: nobody has to write the error-prone parts by hand. The output ends with `oldest first: [Nami (20), Chopper (15), Usopp (12)]`*. Book: Section 13.6.2.*

Python Sorts by Key

Listing 13.15 `age_sort.py`: a key function with `attrgetter`, `reverse=True`, and a two-argument comparator adapted by `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)
```

A **key function** in Listing 13.15 looks at *one* element; Python computes each key once and compares keys with `<`. A key **cannot** break Equation (13.3) or Equation (13.4), and it is faster.

Both sorts are **stable**: equal keys keep their order, so you can sort by first name, then stably by last name.

`(a > b) - (a < b)` is Python's idiom for the sign of a comparison, because `True` and `False` count as 1 and 0. Book: Section 13.6.2.

A compareTo That Only Looks Right

Listing 13.16 `NameOrder.java`, `BrokenName.compareTo`: a `switch` on −1, 0 and 1.

```
        @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;
        }
```

In Listing 13.16, `"Luffy".compareTo("Garp")` is **5** ('L' − 'G' = 76 − 71), not 1. Almost every pair falls through to `Integer.MIN_VALUE`, "less", whatever the order: $a<b$ *and* $b<a$.

Book: Section 13.6.3.

The Sort Just Reverses the Input

```
"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.
```

Java's Timsort looks for a strictly descending **run** and reverses it. A comparison that always says "less" makes the whole array one run, so the result is the **input reversed**, and differs between Case 1 and Case 2.

> ⚠️ **WARNING.** The telltale sign of a broken comparison: **the result depends on the input order**. On large arrays Java may throw `Comparison method violates its general contract!`

Python's sort is the original Timsort and does exactly the same with the same broken comparison. Book: Section 13.6.3.

Field by Field, Done Right

Listing 13.17 `NameOrder.java`: a record that passes the sign on, a last-then-first comparator, and the fragile glued-string comparator.

```
// 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);
    }

    @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());
```

In Listing 13.17, `"Monkey D.Luffy"` versus `"MonkeyZed"`: a space (32) meets `Z` (90), so gluing puts “Monkey D.” first, although `"Monkey"` is a prefix and must come first. Field by field never has this problem.

Python: `@dataclass(frozen=True, order=True)` *compares fields in declaration order, and a tuple key* `lambda n: (n.last_name, n.first_name)` *is the spelling of* `comparing(...).thenComparing(...)`*. Book: Section 13.6.3 and Listing 13.12.*

How Deep Does a Copy Go?

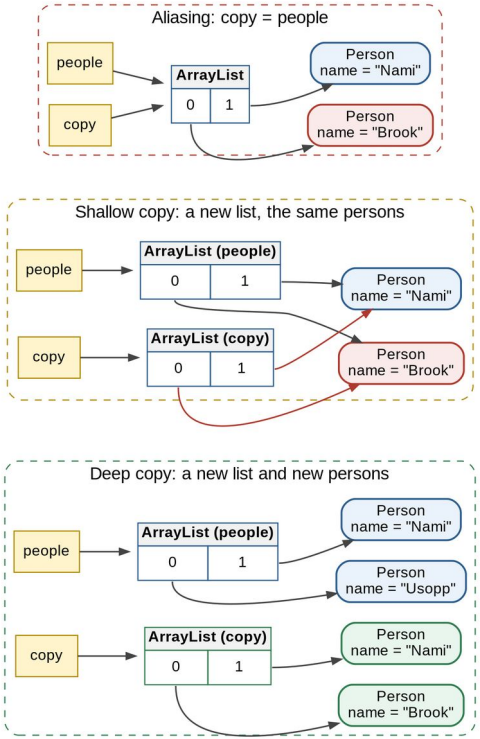


Figure 13.3 Three kinds of "copy" of a list of persons. Aliasing: two variables, one list. Shallow copy: two lists, but their slots refer to the same `Person` objects, so renaming through one list is visible through the other. Deep copy: two lists and separate `Person` objects.

> **DEFINITION. Aliasing**: two references to one object. **Shallow copy**: a new collection holding the *same element references*. **Deep copy**: the collection *and* every element copied, sharing nothing mutable.

Rename Usopp to Brook in the copy of Figure 13.3: the original shows `[Brook, Nami]` for aliasing and shallow copies, and `[Usopp, Nami]` only for a deep copy. Book: Section 13.7 and Figure 13.3.

Shallow and Deep in Java

Listing 13.18 `CopyDemo.java`: a shallow copy and a deep copy of a list of `Person` objects.

```
    // 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;
    }
```

In Listing 13.18, `Collections.copy` after the constructor is harmless but redundant, worth noticing in older code. Java has no general deep copy: each class decides, usually with a copy constructor. Book: Section 13.7.

Three Experiments, Three Answers

```
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]]
```

Python: `list(people)` (or `people.copy()`, `people[:]`, `copy.copy`) is shallow; `copy.deepcopy(people)` copies everything, handling shared references and cycles.

✔️ **KEY INSIGHT.** With **immutable** elements, shallow and deep copies are indistinguishable, and the cheap one is enough.

Book: Section 13.7 and Listing 13.14.

Immutability from Zero

> **DEFINITION.** An object is **immutable** if its state cannot change after it has been created; otherwise it is **mutable**. A *variable* that refers to an immutable object can still be made to refer to another object.

Immutable: `String`, `Integer`, `LocalDate`; Python `int`, `str`, `tuple`, `frozenset`. Mutable: `ArrayList`, `HashMap`, `StringBuilder`, `list`, `dict`, `set`, every class with a setter.

Four benefits. 1. Good **keys**. 2. **Safe arguments**: a method cannot damage them. 3. Shared between **threads** with no locks. 4. **Simple**: one state for life, invariants kept for free.

"Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code." Bank accounts and washing machines stay mutable: they have identity and a life story. Book: Sections 13.8 and 13.8.1.

A Method That Sorts Your List

Listing 13.19 `MedianDemo.java`: a median that sorts its caller's list, and one that sorts a copy.

```java
    // 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);
    }
```

```
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
```

In Listing 13.19 the careless version returns the right median, 44.5, and destroys the week: nobody can tell which day was Monday. An unmodifiable view turns the hidden side effect into a loud `UnsupportedOperationException`*. Python:* `values.sort()` *versus* `sorted(values)`*. Book: Section 13.8.1.*

final Is Not Immutable

Listing 13.20 `ImmutableCollections.java`, blocks 2 and 3: an unmodifiable **view** and an unmodifiable **copy**.

```
        // 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"));
```

```
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
```

*Block 1 adds `"DC"` to a `final` list without complaint. In Listing 13.20, a **view** is read-only but shows later changes behind it; a **copy** is a snapshot nothing can change. `List.of`, `Set.of`, `Map.of` reject `null`, and all of these are **shallow**. Book: Section 13.8.2.*

Tools for Immutability

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

Python in Table 13.3: `typing.Final` is only a hint for checkers; `MappingProxyType` is a read-only view of a dict; `tuple` and `frozenset` are immutable copies and can be dictionary keys, while a `list` cannot.

Book: Section 13.8.2 and Table 13.3.

Five Rules for an Immutable Class

1. **No mutators**: no setters, no `add`, no `increment`.
2. **The class cannot be extended** (`final`), or a subclass could add mutable state.
3. **All fields `private final`**, so the compiler guarantees they are assigned once.
4. **Guard every mutable component**: a **defensive copy** on the way in, never the original on the way out.
5. **Build the object completely in the constructor**; do not let `this` escape. To "change" a value, return a new one: `withX(5)`, `toUpperCase()`, `plusDays(1)`.

Book: Section 13.8.3.

A Leak, and a Record That Copies

Listing 13.21 `ImmutableClasses.java`, `LeakyCustomer`: it keeps the caller's list and hands out its own.

```
    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
        }
    }
```

Listing 13.22 `ImmutableClasses.java`, the record `Customer`: the compact constructor validates the name and copies the list.

```
    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
        }
    }
```

Listing 13.21 leaked twice: the caller's later `add` and a stranger's `add` through the getter both landed inside. A ***record*** *as in Listing 13.22 is `final`, with `private final` fields, accessors, `equals`, `hashCode` and `toString`. The one thing it cannot do for you is the defensive copy. Book: Section 13.8.3.*

Frozen Dataclasses

Listing 13.23 `immutable_classes.py`: a frozen dataclass whose `__post_init__` makes the defensive copy.

```python
@dataclass(frozen=True)
class Customer:
    name: str
    phones: tuple

    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))
```

In Listing 13.23, `frozen=True` makes assignment raise `FrozenInstanceError`; `dataclasses.replace(p, x=5)` is the `withX` pattern; `typing.NamedTuple` is another immutable value type.

> ⚠ **WARNING.** Python's immutability is a strong convention: `object.__setattr__` works from anywhere. It protects against *mistakes*, not tampering; so does Java's `final`.

Book: Section 13.8.3.

Value Objects and Hashing

DEFINITION. A **value object** is completely described by its fields: equal fields mean interchangeable objects, and it is (almost always) immutable. An **entity** has an identity that persists while its fields change.

Money, dates, weights, colors: values. Customers, machines, accounts: entities. The hash contract and the bucket choice:

$$a.\text{equals}(b) \Rightarrow h(a) = h(b) \tag{13.6}$$

$$i = h(k) \bmod m \tag{13.7}$$

A hash table searches only bucket i of Equation (13.7); if Equation (13.6) failed, two equal keys could land in different buckets. Book: Section 13.9.

A Key That Changes

Listing 13.24 `MutableKey.java`: the mutable `Spot(1, 2)` is stored in a map and a set, then its `row` is changed.

```
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());

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());
```

```
hashCode before: 994
hashCode after:  1056
rack.get(spot):              null
rack.get(new Spot(1, 2)):    null
rack.containsKey(spot):      false
```

`Objects.hash(row, col)` *is* $31 \cdot (31 + \text{row}) + \text{col}$*: 994 before, 1056 after. Book: Section 13.9.*

The Bag Is in the Rack, Unreachable

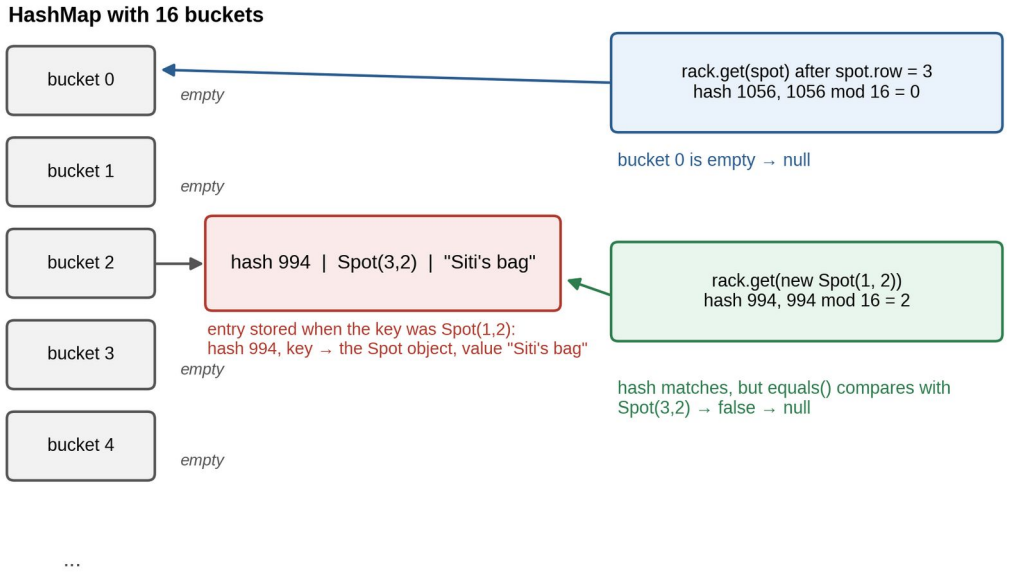


Figure 13.4 A mutated key is lost. The entry was filed in bucket $994 \bmod 16 = 2$; the changed key's hash sends lookups to bucket $1056 \bmod 16 = 0$, which is empty, while a lookup with the old value finds the right bucket but fails the `equals` test.

In Figure 13.4, `rack.get(spot)` looks in bucket 0: empty. `rack.get(new Spot(1, 2))` finds bucket 2 but fails `equals`. The set now holds **two equal elements**.

In the output of Listing 13.24, the entry is still printed by walking all buckets, yet no lookup can ever find it: a memory leak and a lost bag with one assignment. Book: Section 13.9 and Figure 13.4.

The Cure Is Immutability

Listing 13.25 `mutable_key.py`, the class `Shelf`: `frozen` plus the generated `__eq__` gives a safe, hashable key.

```
@dataclass(frozen=True)
class Shelf:                              # frozen + eq  =>  a safe, hashable key
    row: int
    col: int
```

Listing 13.26 `mutable_key.py`: to "move" a bag, remove it under the old key and store it under a new one; a list is refused as a key.

```
    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)
```

✔️ **KEY INSIGHT.** Use as keys only objects whose equality **cannot change**: strings, numbers, dates, enums, records, frozen dataclasses, tuples of these. An entity is keyed by an unchanging id.

Listing 13.25 is the cure; Listing 13.26 shows Python going one step further: `list`, `dict` and `set` are unhashable, so `{[1, 2]: "bag"}` fails at once with `TypeError`. Book: Section 13.9.

Filter, Map, Reduce

"Revenue from loads of at least 4 kg at Rp7,000 per kilogram": **filter** $w \geq 4$, **map** $w \mapsto 7000\,w$, **reduce** with $+$ from 0:

$$\text{reduce}(f, e, [x_1, x_2, \ldots, x_n]) = f(\cdots f(f(e, x_1), x_2) \cdots, x_n) \tag{13.8}$$

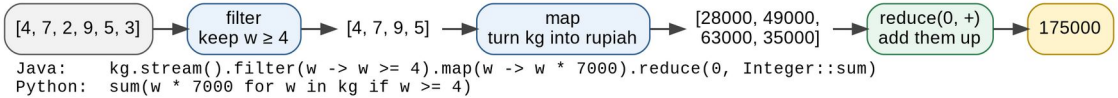


Figure 13.5 A filter–map–reduce pipeline. The source loads flow through a filter and a map into a reduction that folds them into one number; the same pipeline in one line of Java and one line of Python.

For $f = +$ and $e = 0$, Equation (13.8) is $x_1 + \cdots + x_n$; for $f = \max$, the largest element. Figure 13.5 is the pipeline in one line of each language. Book: Section 13.10.

Streams in Java

Listing 13.27 `StreamsDemo.java`: filter, map and reduce as a stream, and laziness made visible with `peek`.

```
        // 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));
```

```
loop:   175000
stream: 175000
pipeline built, nothing printed yet
 saw 4 saw 7 -> first heavy load: 7
```

*A stream stores nothing: source, intermediate operations, one terminal operation. In Listing 13.27, `findFirst` stops at 7; 2, 9, 5 and 3 are never examined. A stream is **single-use**; collectors such as `groupingBy` and `joining` turn it back into a map or a string. Book: Section 13.10.*

Comprehensions in Python

Listing 13.28 `streams_demo.py`: a loop, a generator expression, `map`/`filter`/`reduce`, and laziness.

```
    # 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))
```

```
comp:   175000
reduce: 175000
pipeline built, nothing printed yet
 saw 4 saw 7 -> first heavy load: 7
```

*In Listing 13.28, brackets build a list **eagerly**; parentheses give a **lazy** generator. `groupby` needs sorted input; `count`, `islice` and `accumulate` complete the toolbox. Clarity, not fashion, decides between a loop and a pipeline. Book: Section 13.10 and Table 13.4.*

The Laundry Ledger

Weights in **tenths of a kilogram**, money in whole rupiah in a `long`, never a `double`. Chargeable weight:

$$\hat{w} = 5 \left\lceil \frac{\max(w, w_{\min})}{5} \right\rceil \tag{13.9}$$

EXAMPLE. 4.2 kg of Wash & Fold by Equation (13.9): $\max(42, 30) = 42$, $\lceil 42/5 \rceil = 9$, $\hat{w} = 45$, price $7000 \cdot 45/10 = 31{,}500$. In integers, $\lceil a/5 \rceil$ is `(a + 4) / 5`.

The price rules are those of Chapter 7: `WF` Rp7,000 (3 kg minimum), `IR` Rp5,000 (2 kg), `WDI` Rp10,000 (3 kg), members 10% off, rounded down. Book: Section 13.11.

The Design

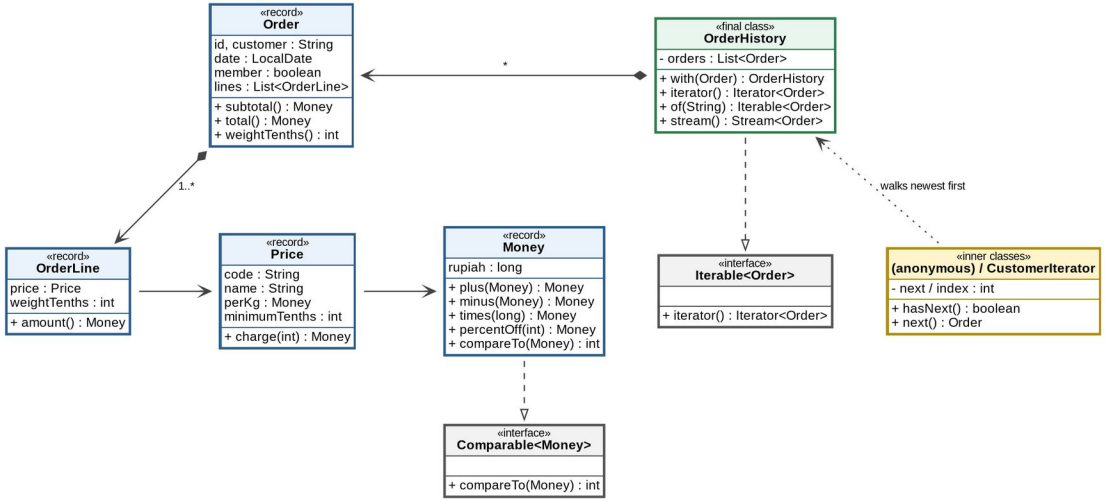


Figure 13.6 UML class diagram of the laundry ledger. `Money`, `Price`, `OrderLine` and `Order` are records (immutable values); `OrderHistory` holds an unmodifiable list of orders, implements `Iterable<Order>`, and hands out two kinds of iterator.

In Figure 13.6, four records are values: `Money`, `Price`, `OrderLine`, `Order`. `OrderHistory` is an immutable, append-only `Iterable<Order>` with two kinds of iterator.

Book: Section 13.11 and Figure 13.6.

Money: A Value Object

Listing 13.29 `LaundryLedger.java`, the record `Money`: its compact constructor forbids negative amounts, and 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
    }
```

In Listing 13.29, `minus` needs no check: the constructor refuses -2000*. `addExact` and `multiplyExact` throw instead of wrapping as in Equation (13.5). `Money` implements `Comparable<Money>`. Book: Section 13.11.*

Prices and Frozen Orders

Listing 13.30 `LaundryLedger.java`, the record `Price`: Equation (13.9) in code.

```
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);
    }
```

Listing 13.31 `LaundryLedger.java`, the record `Order`: a defensive, unmodifiable copy of its lines, and a total computed with a small stream.

```
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);
    }

    Money total() {
        return member ? subtotal().percentOff(10) : subtotal();
    }
```

Listing 13.30 charges a line; Listing 13.31 ensures an order, once built, can never gain or lose a line: `first.lines().add(...)` *throws. Book: Section 13.11.*

An Append-Only History

Listing 13.32 `LaundryLedger.java`, `OrderHistory`: `with(order)` returns a **new** history one order longer.

```
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();
    }
```

Listing 13.32 is the `withX` pattern applied to a collection: the report, the counter screen and a backup can share one history without any risk. Book: Section 13.11.

Newest First, and One Customer

Listing 13.33 `LaundryLedger.java`: `iterator()` returns a newest-first anonymous iterator.

```
    @Override
    public Iterator<Order> iterator() {
        return new Iterator<>() {
            private int next = orders.size() - 1;

            @Override
            public boolean hasNext() {
                return next >= 0;
            }

            @Override
            public Order next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                return orders.get(next--);
            }
        };
    }
```

Listing 13.33 makes the counter screen one for-each loop. `of(customer)` *returns an* `Iterable` *written as a lambda,* `() -> new CustomerIterator(customer)`*, which uses* ***look-ahead****:* `skipOthers()` *moves past other customers so that* `hasNext()` *is one comparison. Nothing is copied. Book: Section 13.11.*

What the Report Prints

```
== 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]
```

Revenue per service (Rp341,500) is before discounts; total revenue Rp329,650 is after Siti's three member discounts of Rp4,150, Rp3,500 and Rp4,200. Book: Section 13.11.

The Ledger in Python

Listing 13.34 `laundry_ledger.py`, `OrderHistory`: a tuple, `with_order`, a generator for newest-first, and a generator expression for one customer.

```
    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)
```

`Money` is `@dataclass(frozen=True, order=True)` with `__add__`, `__sub__`, `__mul__`, so `sum(..., ZERO)` works: Equation (13.8) with e = `ZERO`. The whole Java anonymous iterator of Listing 13.33 becomes two lines in Listing 13.34.

The output matches the Java program apart from the printing of lists and exception names. Book: Section 13.11.

Profitina in Practice

Scan results as values. A finished scan is a fact about the past; immutable results make the historical analytics dashboard trustworthy, and the CSV export writes exactly what the chart shows.

Walking large histories lazily. A brand's trend is a newest-first, filtered walk over growing scan history, like the `OrderHistory` iterators.

Reports as pipelines. Averages per week and most-improved signals are filter–map–reduce comprehensions over immutable results.

Personal data. Under UU PDP No. 27/2022, an export receives an immutable snapshot. The dashboard explains; the user decides.

Book: Section 13.12.

Common Mistakes

1. `next()` **twice in one loop body. 2. Changing a collection's size inside for-each** (Table 13.2). **3. Walking a Python iterator twice**: empty the second time, no error.

4. `compareTo` **that handles only** $-1, 0, 1$ (Listing 13.16). **5. Comparing by subtraction** (Equation (13.5)). **6. Treating a shallow copy as independent.**

7. Believing `final` **makes an object immutable. 8. Confusing a view with a copy. 9. No defensive copy** (`LeakyCustomer`).

10. Mutating a key while it is stored (Figure 13.4). **11. Side effects inside a stream or comprehension. 12. Reusing a stream.**

Each mistake has its cure in the book, next to the program that shows it. Book: Section 13.13.

Summary — Five Things

1. **Iterators walk; iterables produce iterators.** `hasNext`/`next` (Equation (13.1)), `iter`/`next`/`StopIteration`; a generator writes the iterator for you, lazily. An iterator is used once.
2. **Never change a collection's size while walking it.** Java detects it with `modCount` (Equation (13.2)), on a best-effort basis; Python lists skip silently.
3. **Comparisons must be antisymmetric and transitive**; only the sign matters; never subtract. Python sorts by key; both sorts are stable.
4. **Immutable objects** are safe keys, safe arguments, safe to share, and keep their invariants. `final` fixes a reference; records and frozen dataclasses need defensive copies of mutable parts.
5. **Value objects** hash by their fields (Equation (13.6)); keys must not change. **Streams and comprehensions** are lazy, single-use filter–map–reduce pipelines.

Book: Section 13.14.

Next Time

Chapter 14 — Threads, Races, Deadlock, and Livelock. Several pieces of code running at the same time, and the three ways that goes wrong.

Today's third benefit of immutability, *shared between threads with no locks*, becomes a necessity: mutable shared state is where races come from.

Homework. Exercises 13.1–13.13; start with 13.4 (predicting `ConcurrentModificationException` with Equation (13.1) and Equation (13.2)) and 13.8 (finding the leaks in `Voucher`).

Solutions are in Section 13.16; try each exercise before reading its solution. Book: Sections 13.15 and 13.16.

Lecture 14 — Threads, Races, Deadlock, and Livelock

Object-Oriented Programming (EF234302)

Chapter 14 of the book, in miniature, for the classroom.

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*.

Lecturer: Irfan Subakti — yifana@gmail.com

A concurrent program can print something different every time you run it; today we show real runs and say exactly what varies and what must never vary. Book: Chapter 14, opening paragraph.

Agenda

1. **Threads.** What a thread is, concurrency versus parallelism, creating and stopping threads in Java and Python, and the life of a thread.
2. **Speed.** Amdahl's law, and Python's Global Interpreter Lock: when threads help and when they do not.
3. **Races.** A shared counter that loses updates, why `c++` is not one step, and the cures: `synchronized`, locks, atomics.
4. **Waiting.** `wait`/`notifyAll`, `Condition`, and blocking queues for producers and consumers.
5. **Liveness.** Deadlock, its four conditions and its prevention; livelock and starvation; and a laundry of three machine threads.

The pages follow the book in order; the exercises at the end of the chapter are the homework. Book: Sections 14.2 to 14.14.

Where We Left Off

Every program so far did **one thing at a time**. Chapter 10 was the quiet exception: a Swing or Tkinter window kept repainting while our handler ran, because *someone else's* thread ran the event loop.

Chapter 13 made objects **immutable**, which turns out to be one of our best defenses today.

Profitina Laundry buys two more washing machines, but its software still starts a wash, waits, records the payment, and only then starts the next: two machines stand idle.

> ✔️ **KEY INSIGHT.** Threads are easy to *start* and hard to *coordinate*. Almost every concurrency bug comes from one situation: two threads touch the same data, and at least one of them changes it.

Book: Section 14.1.

The Laundry Is a Computer

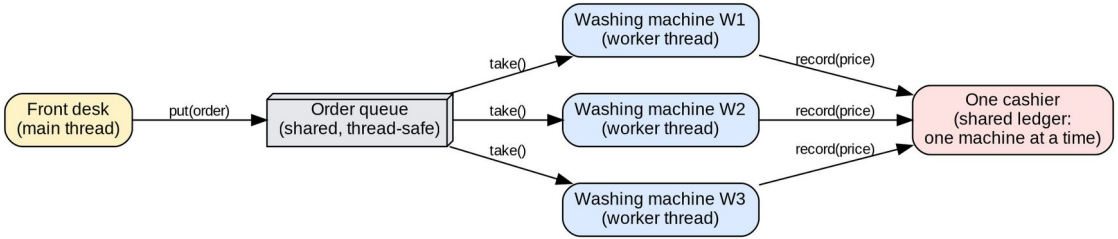


Figure 14.1 Profitina Laundry as a concurrent system. The front desk puts orders into one shared queue; three washing machines, each a worker thread, take orders from it; all three pay into one ledger, which accepts one machine at a time.

In Figure 14.1, the three machines wash **simultaneously**; the one cashier writes **one payment at a time**. The picture is also, almost word for word, the program of Section 14.14.

Book: Section 14.2 and Figure 14.1.

Processes and Threads

> **DEFINITION.** A **process** is a running program with its own memory, open files and at least one thread. A **thread** is one independent sequence of instructions inside a process. Each thread has its own *call stack*; all threads of a process share the same *heap*.

A machine's local variables (which order, how many kilograms) are its stack; the basket of orders and the cashier's book are shared objects on the heap. Sharing makes threads **useful** and **dangerous**.

Why threads? Responsiveness (a window that does not freeze), waiting in parallel (one thread per client), using every core, and simpler models (one loop per activity).

Every program has at least one thread: `main` *in Java,* `MainThread` *in Python. Book: Sections 14.2 and 14.2.2.*

Concurrency versus Parallelism

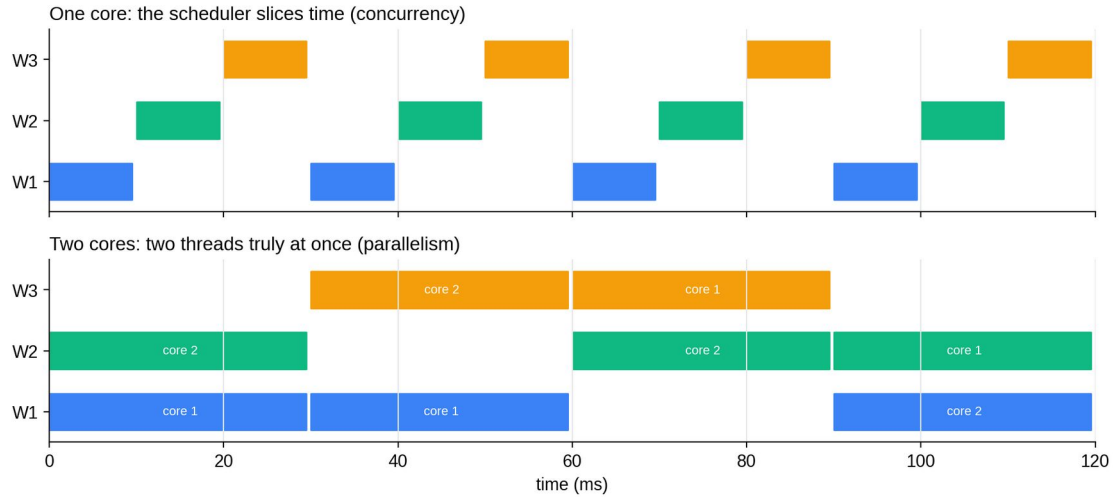


Figure 14.2 Concurrency versus parallelism. Top: one core; the scheduler gives each thread a short time slice in turn, so all three make progress but only one runs at any instant. Bottom: two cores; two threads really run at once, and the scheduler moves threads between cores as it likes.

You do not decide the order: the scheduler does, differently on every run. **A thread can be paused anywhere**, even in the middle of `count++`.

In Figure 14.2, concurrency means several tasks in progress; parallelism means several executing at the same instant on different cores. Book: Section 14.2.1 and Figure 14.2.

A Parent Who Cannot Get Back Control

```
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++));
        }
    }
}
Child one = new Child("Luffy");
one.begin();
Child two = new Child("Ace");
two.begin();
```

`one.begin()` never returns: the parent's only thread is busy inside the child's infinite loop. The second child is never created, and the first can never be stopped.

We want `begin` to start the counting somewhere else and give control back at once. That is exactly what a thread does. Book: Section 14.3.

The Child as a Runnable

Listing 14.1 `ParentChild.java`, `Child.run()`: print, sleep a random time, and end when interrupted.

```
        @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");
            }
        }
```

In Listing 14.1, `Thread.sleep` throws `InterruptedException` when another thread calls `interrupt()` on the sleeper; the exception leaves the loop and `run()` returns. Book: Section 14.3.1.

The Parent Starts, Sleeps, Interrupts

Listing 14.2 `ParentChild.java`, `main`: `start()` returns at once; `interrupt()` asks, `join()` waits.

```
        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();
        }
        threadOne.join();  // wait until each child has really finished
        threadTwo.join();
        System.out.println("Parent ended");
```

Listing 14.2: we never call `run()` ourselves; `start()` asks the JVM for a new thread, which calls `run()`. The `finally` block asks the children to stop even if the parent's sleep were cut short. Book: Section 14.3.1.

One Real Run

```
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
```

The lines of Luffy and Ace **interleave**; Ace's short first sleeps let him count to 5 before Luffy wakes. `Parent ended` comes last because of `join()`.

> ⚠️ **WARNING.** An interrupt is a *request*, not a command: it sets a flag and wakes a sleeper. The deprecated `Thread.stop()` left objects half-changed.

The seeds fix the sleep times; what varies is the fine timing near a millisecond. The pattern is the same on every run; the details are not. Book: Section 14.3.1.

Python: An Event Instead of an Interrupt

Listing 14.3 `parent_child.py`, `Child.run()`: `stop.wait(timeout)` sleeps like `time.sleep`, but wakes early when the parent calls `stop.set()`.

```
    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
```

Python threads cannot be interrupted. In Listing 14.3 a shared `threading.Event` replaces `interrupt()`: `set()` raises the flag, `wait(t)` sleeps until the flag is raised or `t` passes.

Pass the bound method, `target=child.run`, never `target=child.run()`, which would run the endless loop in the parent's thread. Book: Section 14.3.1.

The Visibility Trap

Listing 14.4 `StopFlag.java`, first half: a thread spinning on a plain `boolean` flag.

```
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());
```

Listing 14.4 prints `still running 1 s later? true`: the spinner **never stops**. The JIT compiler may read `plainStop` once, keep it in a register, and turn the loop into `while (true)`.

The same loop on a `volatile` flag stops at once: every read of a volatile field sees the latest write. Book: Section 14.3.2.

Visibility and Happens-Before

> **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 visible only through a *happens-before* relation: a `volatile` write and read, the same lock, or `start`/`join`.

In CPython the plain attribute *does* become visible (the GIL), but that is a fact about today's interpreter, not a promise; and a spinning loop wastes a core.

Idioms. Java: a `volatile` flag, or interrupts. Python: `threading.Event` with `set()`, `is_set()` and `wait(timeout)`.

Book: Section 14.3.2.

Four Ways to Create a Thread

1. **Extend `Thread`** and override `run()`: the oldest style; the class cannot extend anything else, and it mixes *what* with *how*.
2. **Implement `Runnable`** and hand it to `new Thread(r, name)`: *what* to do is separate from *how* to run it. The recommended design.
3. **A lambda**: `Runnable` is a functional interface, so `new Thread(() -> ..., "iron-1")` works.
4. **An `ExecutorService`**: a pool of reusable worker threads fed with tasks.

Java 21 also offers virtual threads, `Thread.ofVirtual().start(task)`; races, locks and deadlock work the same way with them. Book: Section 14.4.

A Pool and Its Futures

Listing 14.5 `CreateThreads.java`, way 4: a fixed pool of two threads computes four prices as `Callable` tasks.

```
        // 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);
```

```
pool finished: true
total of 4 loads = Rp147000
```

In Listing 14.5, `submit` returns a `Future`, a ticket for a result; `get()` waits for it. $21 \times 7000 = 147,000$. *`shutdown()` accepts no new tasks and lets the old ones finish. Book: Section 14.4.*

run() Is Not start()

Listing 14.6 `CreateThreads.java`: calling `run()` executes the task in the current thread; only `start()` creates a new one.

```
        // 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();
```

```
run() executed by: main
run() executed by: helper
```

Listing 14.6 shows the most common beginner's mistake: it gives correct answers, slowly and without any concurrency, which is why it survives so long.

> ⚠ **WARNING.** A thread can be started only once: a second `start()` throws `IllegalThreadStateException`. To run the work again, create a new `Thread` or submit the task again.

Book: Section 14.4.

The Thread Vocabulary

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()`

Python in Table 14.1: a target function is the `Runnable`; `ThreadPoolExecutor` in a `with` block shuts down and waits. In a loop, write `lambda kg=kg: kg * 7000`, or every lambda sees the last `kg`.

*A program ends when its last **non-daemon** thread ends; daemons are background helpers that may be abandoned. Book: Section 14.4 and Table 14.1.*

The Life of a Thread

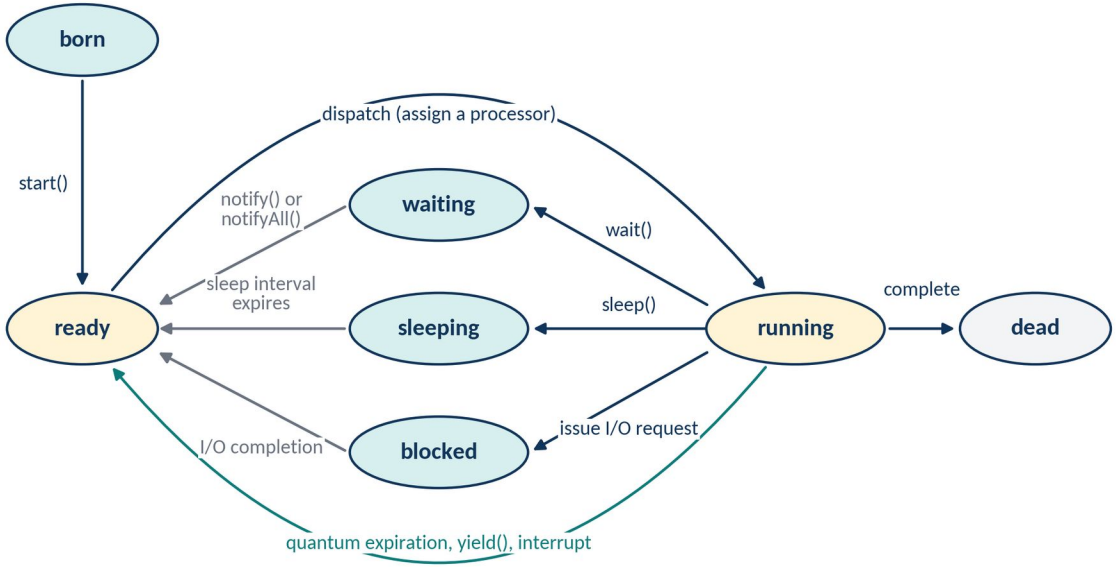


Figure 14.3 The classic life cycle of a Java thread. A thread is born, becomes ready when started, runs when the scheduler dispatches it, and can leave the running state to wait, sleep or block on input and output before it becomes ready again or dies.

Figure 14.3 separates *ready* (able to run) from *running* (on a core now). Java does not make that distinction visible: the scheduler switches thousands of times per second.

Book: Section 14.5 and Figure 14.3.

The Six Values of Thread.State

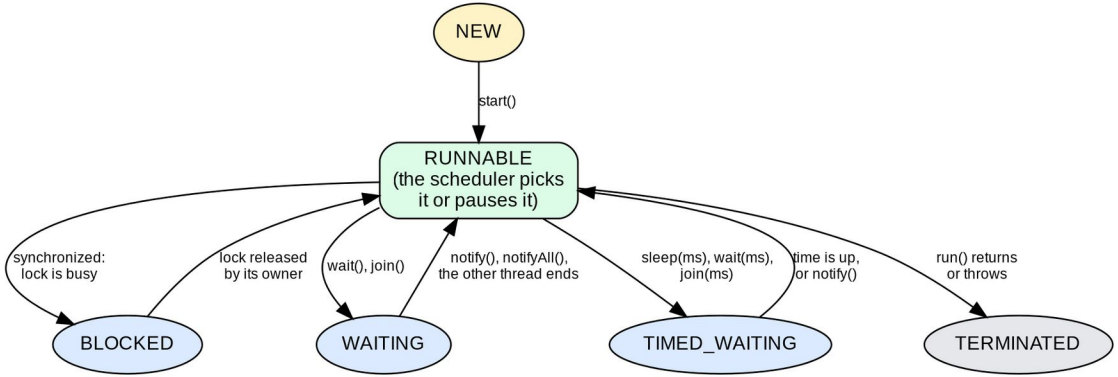


Figure 14.4 The six values of `Thread.State` and the events that move a thread between them. `RUNNABLE` covers both "ready" and "running" of Figure 14.3; `BLOCKED` means waiting for a lock, `WAITING` means waiting without a time limit, and `TIMED_WAITING` means waiting with one.

In Figure 14.4: **`NEW`** not started; **`RUNNABLE`** running or ready; **`BLOCKED`** waiting for a lock; **`WAITING`** waiting with no limit (`wait()`, `join()`); **`TIMED_WAITING`** with a limit (`sleep(ms)`); **`TERMINATED`** finished for good.

Book: Section 14.5 and Figure 14.4.

Catching a Thread in BLOCKED

Listing 14.7 `ThreadStates.java`: `main` holds the lock, so the thread `blocked` cannot enter and is `BLOCKED`.

```
        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();
        }
```

```
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
```

The other observations of Listing 14.7 are arranged the same way, with a short `sleep(100)` in `main`. Python has no state enumeration: `is_alive()`, `ident`, `Lock.locked()` and `threading.enumerate()`. Book: Section 14.5.

Do Threads Make It Faster?

Listing 14.8 `SpeedupDemo.java`: the primes below three million counted by two threads, one per half.

```
        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();
```

```
primes below 3000000 : 216816 and 216816
one thread           :  623 ms
two threads          :  602 ms
```

Amdahl's law: if a fraction f of the work splits over p cores,

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}} \tag{14.1}$$

With $f = 0.9$, $p = 4$: $S(4) = 1/0.325 \approx 3.08$, and Equation (14.1) can never exceed $1/(1-f) = 10$. Listing 14.8 ran on a busy machine; the result 216,816 never varies. Book: Section 14.6.

Python's Global Interpreter Lock

DEFINITION. The **GIL** is a lock inside CPython. A thread must hold it to execute Python bytecode, and only one thread can hold it at a time; it is released regularly (every 5 ms) and whenever a thread blocks.

Listing 14.9 `gil_demo.py`: prime counting in threads and processes, and four half-second waits.

```
    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)))
    with ThreadPoolExecutor(4) as pool:
        timed("four 0.5 s waits, four threads", lambda: sum(pool.map(wait_for_device, waits)))
```

```
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
```

In Listing 14.9, four waits drop from 2.00 s to 0.50 s; two computing threads gain little. Book: Sections 14.6 and 14.6.1.

When Python Threads Help

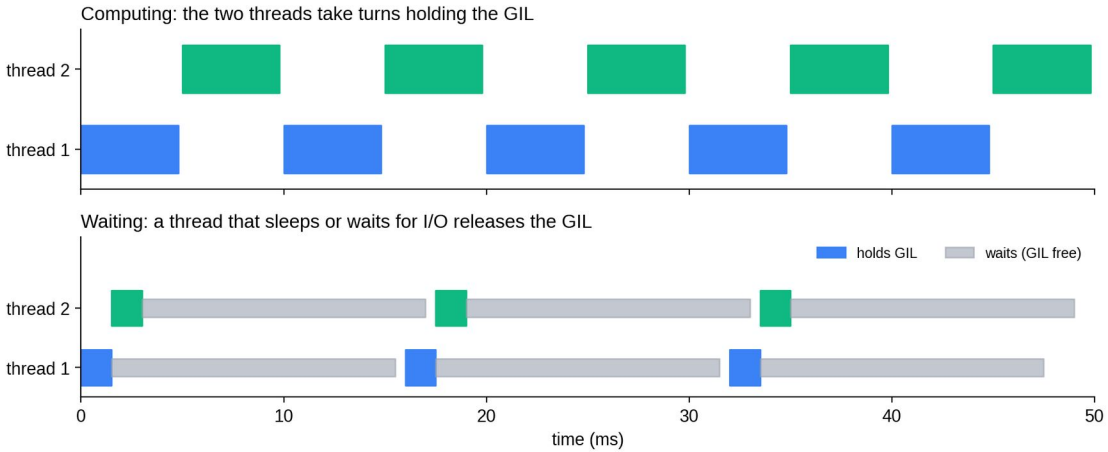


Figure 14.5 The GIL in pictures. Top: two computing threads take turns holding the GIL, so they never compute at the same moment. Bottom: threads that mostly wait release the GIL while waiting, so their waits overlap.

In Figure 14.5: **I/O-bound** work, threads help. **CPU-bound** pure Python, use processes (`ProcessPoolExecutor`). C libraries that release the GIL, threads help.

*The GIL does **not** make your code thread-safe. Python 3.13 added an experimental free-threaded build (PEP 703), officially supported in 3.14 (PEP 779). Book: Section 14.6.1 and Figure 14.5.*

Three Children, One Counter

Listing 14.10 `RaceDemo.java`, `Child.run()`: after a common start signal, increment (and perhaps decrement) a shared counter a million times.

```
        @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();
                }
            }
        }
```

```
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
```

In Listing 14.10, the counter that "must be 3,000,000" lost almost two million increments in round 3, and was perfect in rounds 4 and 5. No exception, no warning. Book: Section 14.7.

Race Conditions

> **DEFINITION.** A **race condition** is a bug in which the result depends on the relative timing of threads. The most common kind is the **lost update**: two threads read the same old value, and the second write overwrites the first.

```
  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
```

`getfield` **reads**, `iconst_1` and `iadd` **add** on a private copy, `putfield` **writes**. The scheduler may pause a thread between any two.

What varies: the number of lost updates. What never varies: the increment-only counter can only be too small. Book: Sections 14.7 and 14.7.1.

A Lost Update, Step by Step

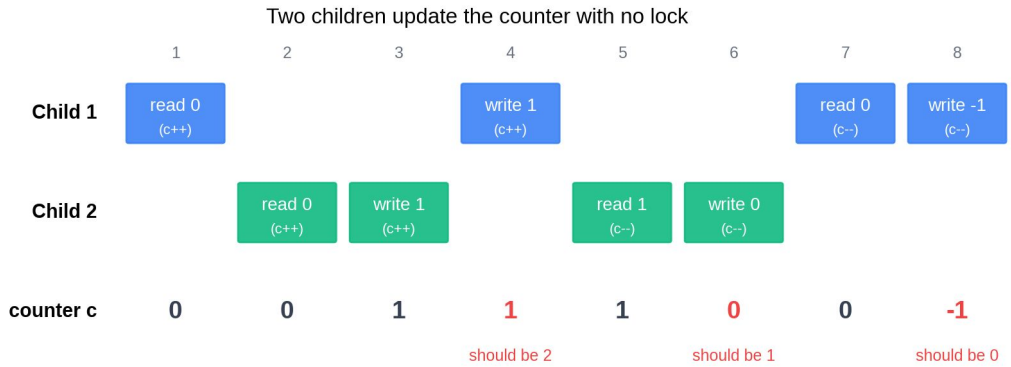


Figure 14.6 A lost update, step by step. Child 1 reads 0; Child 2 reads 0 and writes 1; Child 1 writes its own 1 over it. Two increments have happened and the counter says 1. Two more steps later a decrement is lost as well, and the counter ends at -1 instead of 0.

In Figure 14.6: Child 1 reads 0; Child 2 reads 0 and writes 1; Child 1 writes its own 1 over it. Two increments, and the counter says 1. Each thread did exactly what its code said; the error lives in the **interleaving**.

Book: Section 14.7.1 and Figure 14.6.

Counting the Interleavings

A thread of m steps and one of n steps can interleave in

$$\binom{m+n}{m} = \frac{(m+n)!}{m!\,n!} \tag{14.2}$$

ways. Two increments, $m=n=3$: 20 interleavings, only 2 correct.

With T threads of N increments each, the updates lost are

$$L = T \cdot N - c_{\text{end}} \tag{14.3}$$

Round 1: $L = 3,000,000 - 2,423,058 = 576,942$.

⚠ **WARNING.** The counter in Listing 14.10 is `volatile`, and it still loses updates: **`volatile` does not make `c++` safe.**

By Equation (14.2), testing can show that a race exists; it can never show that one does not. Equation (14.3) is what the program prints as `lost`. Book: Section 14.7.1.

The Same Race in Python

Listing 14.11 `race_demo.py`, `SlowCounter`: the same three steps, with `time.sleep(0)` between the read and the write.

```
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
```

```
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
```

In Listing 14.11, plain `self.c += 1` happened to lose nothing in CPython 3.11, but its bytecode is still `LOAD_ATTR`, `BINARY_OP`, `STORE_ATTR`; `sleep(0)` only opens the window of the old bug. Book: Section 14.7.2.

Mutual Exclusion

DEFINITION. A **lock** (*mutex*) is an object that at most one thread can hold at a time. A thread **acquires** it before a **critical section** and **releases** it after; a thread that finds it held waits.

```
public synchronized void increment() {
    c++;
}
synchronized (counter) {
    counter.increment();
}
```

Every Java object has an intrinsic lock. **Every access to shared data that some thread changes must hold the same lock**, readers included.

A synchronized method locks `this` and releases it even when it throws; a synchronized block locks any object you name. Book: Section 14.8.

An Explicit Lock and an Atomic

Listing 14.12 `SafeCounters.java`, cure 3: a `ReentrantLock`, always released in `finally`.

```
// Cure 3: an explicit lock, released in finally.
static class LockCounter implements Counter {
    private final ReentrantLock lock = new ReentrantLock();
    private int c = 0;

    public void increment() {
        lock.lock();
        try {
            c++;
        } finally {
            lock.unlock();
        }
    }
```

Listing 14.13 `SafeCounters.java`, cure 4: an `AtomicInteger`; the hardware does read-add-write in one step.

```
// 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();
    }
```

Listing 14.12: `tryLock(time, unit)` can give up, and `new ReentrantLock(true)` is fair. Listing 14.13: a compare-and-swap loop, no lock at all. Book: Section 14.8.

Five Cures, One Answer

```
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)
```

> **DEFINITION.** An operation is **atomic** if no other thread can ever observe it half done. `c++` is not; `incrementAndGet()` is; any sequence becomes atomic for threads that use the same lock.

A lock also fixes **visibility**: releasing it happens-before every later acquisition. Java locks are **reentrant**; Python's `Lock` is not, `RLock` is.

The timings vary and are no benchmark; the totals of 3,000,000 never vary. `LongAdder` spreads the count over cells for heavy contention. Book: Sections 14.8 and 14.8.1.

Locks in Python

Listing 14.14 `safe_counters.py`, `LockedCounter`: `with self._lock:` makes the slow read-modify-write correct.

```
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
```

```
LockedCounter       : should be 6000, is 6000  (0.42 s)
ExplicitLockCounter : should be 6000, is 6000  (0.42 s)
```

In Listing 14.14, `sleep(0)` still hands over the GIL mid-update, but the next thread finds the lock held and waits. `with` is `acquire`, `try`, `finally: release()`; prefer it. Book: Section 14.8.2.

The Synchronization Toolbox

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 never shared cannot race, and immutable data (Chapter 13) needs no lock. Lock what is shared *and* changed, and only that.

Table 14.2 collects every tool of the chapter. Book: Section 14.8.2 and Table 14.2.

Producer, Consumer, and a Ring Buffer

The producer waits while the buffer is full; the consumer waits while it is empty. Five cells, used circularly:

$$w \leftarrow (w+1) \bmod n, r \leftarrow (r+1) \bmod n \tag{14.4}$$

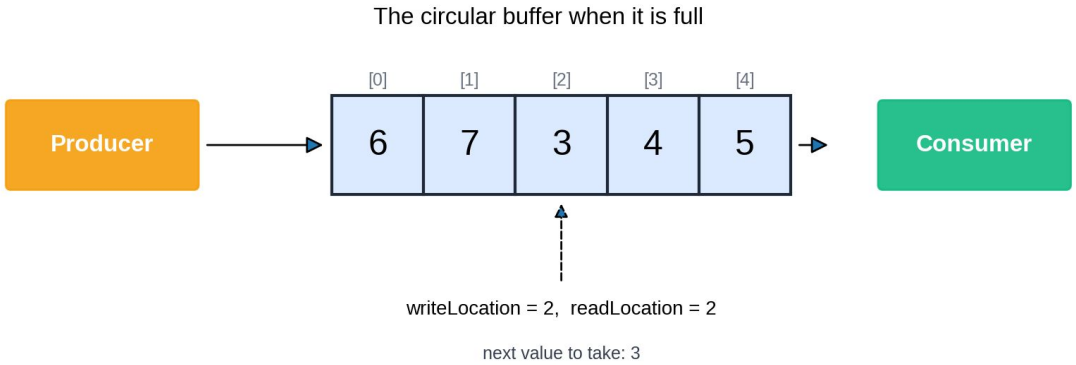


Figure 14.7 The five-cell circular buffer when it is full. Both indexes point to cell 2: the consumer's next value is the 3 stored there, and the producer's next value will go there as soon as the 3 has been taken.

When the indexes of Equation (14.4) meet, the buffer is full or empty; the flags `writeable` and `readable` say which. Figure 14.7 shows the full state. Book: Section 14.9 and Figure 14.7.

wait() and notifyAll()

Listing 14.15 `ProCon.java`, `setSharedInt`: wait while no cell is free, write, move the index, and wake the consumer.

```
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
}
```

*`wait()` releases the lock and sleeps; `notifyAll()` wakes the waiters, who re-acquire the lock before `wait()` returns. A lock plus a waiting room is a **monitor** (Hoare). Book: Section 14.9 and Listing 14.15.*

while, Not if

```
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]
```

`while (!writeable) wait();`: a woken thread must test again; another thread may have taken the cell, and Java permits spurious wake-ups.

`wait()` **releases the lock**, so the consumer can enter `getSharedInt` while the producer waits. `notifyAll()`, not `notify()`, so the right kind of thread is never left asleep.

In the output of Listing 14.15 the values are consumed 1 to 10, each once, and the total is 55 on every run. Book: Section 14.9.

The Exchange in a Window

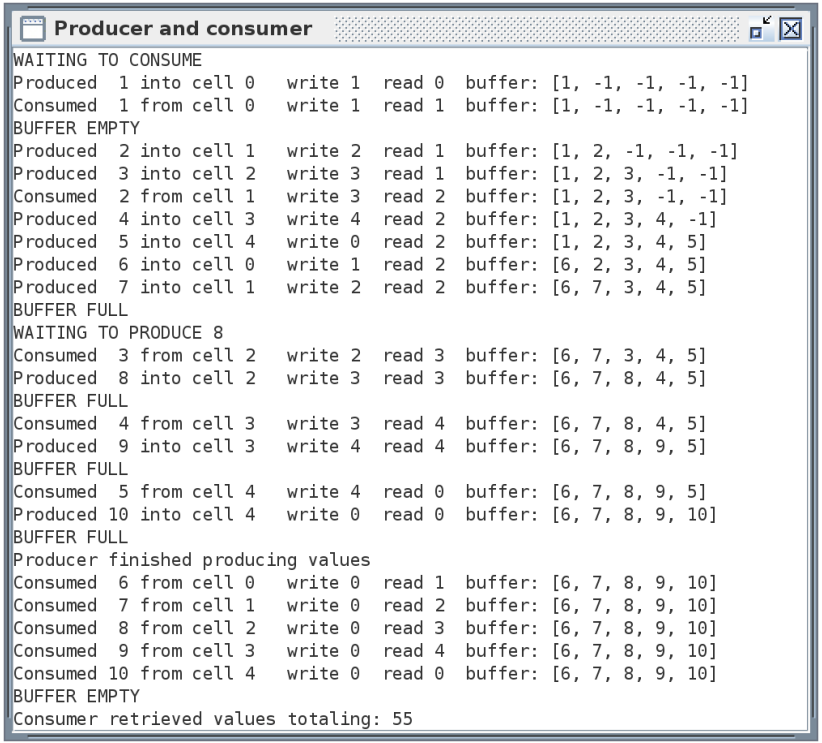


Figure 14.8 The output of `ProCon.java` shown in a Swing text area: the producer writes 1 to 10 into five circular cells, waits when the buffer is full, and the consumer's total is 55.

Figure 14.8: the same messages, the same wrap-around from cell 4 to cell 0, and the same total of 55.

Book: Section 14.9 and Figure 14.8.

Python: threading.Condition

Listing 14.16 `procon.py`, `set_shared_int`: `with self.cond:` takes the lock; `cond.wait()` releases it and waits; `notify_all()` wakes the waiters.

```
    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
```

Listing 14.16 is a line-by-line translation. Java offers the same explicit style: `lock.newCondition()` with `await()` and `signalAll()`. Book: Section 14.9.

Blocking Queues Do the Waiting

Listing 14.17 `QueueDemo.java`, the producer: `put` waits while the `ArrayBlockingQueue` of five cells is full.

```
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();
    }
});
```

Listing 14.18 `QueueDemo.java`, the consumer: `take` waits while the queue is empty, and a **poison pill** ends the loop.

```
    try {
        while (true) {
            int value = buffer.take();  // waits while the queue is empty
            if (value == done) {
                break;
            }
            sum += value;
            count++;
        }
```

Listing 14.17 and Listing 14.18 print `Consumer took 10 values totaling: 55` on every run. After catching `InterruptedException`, restore the flag with `Thread.currentThread().interrupt()`. Book: Section 14.10.

queue.Queue in Python

Listing 14.19 `queue_demo.py`: `put` and `get` wait on a full or empty `queue.Queue(maxsize=5)`; `None` is the poison pill.

```
    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}")
```

✔ **KEY INSIGHT.** A blocking queue replaces *shared data* (which needs locks) with *messages* (which belong to the receiver alone). It is the single most useful concurrency tool.

Internally, the queue of Listing 14.19 is a list protected by a lock and two conditions, "not empty" and "not full". Book: Section 14.10.

The Dining Philosophers

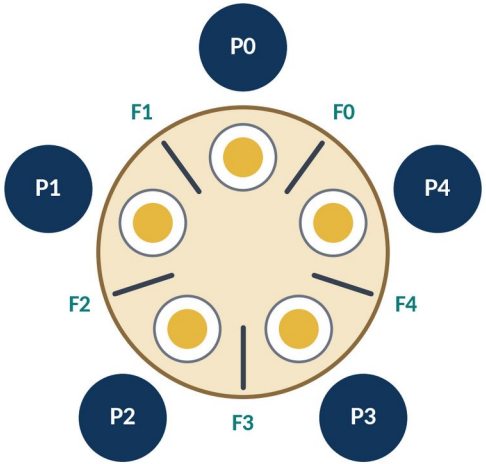


Figure 14.9 The dining philosophers: five philosophers, five plates of spaghetti, and one fork between each pair of neighbors. Philosopher P_i needs forks F_i and $F_{(i+1) \bmod 5}$, as in `Philosopher.java`.

In Figure 14.9, to eat a philosopher needs **both** neighboring forks. The obvious discipline: pick up the left fork, then the right, eat, put both down.

Posed by Dijkstra in 1965 and given its present form by Hoare. Each philosopher is a thread and each fork is a lock. Book: Section 14.11.1 and Figure 14.9.

Left, Then Right

Listing 14.20 `Philosopher.java`, `run()`: think, take the left fork, wait one unit, take the right fork, eat.

```
    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, "<0 ");
                    sleep(UNIT);
                    synchronized (forks[right]) {           // take the right fork
                        show(right, " ", me, "<0>");
                        sleep(UNIT);                        // eat
                        ate(me);
                        show(right, "^", me, "<0 ");        // put the right fork back
                    }
                    show(left, "^", me, "_0_");             // put the left fork back
                }
            }
```

In Listing 14.20, `sleep(UNIT)` while holding the left fork is the dangerous gap: any work done between acquiring two locks. The drawing lock is always taken inside the fork locks. Book: Section 14.11.1.

The Diner Is Locked

```
  <0  ^ _0_ ^ _0_   <0  ^ _0_   32
  <0>   _0_   <0    <0    <0    33
  _0_ ^ _0_   <0    <0    <0>   34
  <0    <0    <0    <0>   _0_   35
  <0    <0    <0    <0    <0    36
The diner is locked.
Meals eaten before the deadlock: [7, 5, 5, 7, 6]
```

`^` a fork on the table, `_0_` no fork, `<0` the left fork only, `<0>` eating. At drawing 36 every philosopher holds a left fork and waits for a right fork the neighbor holds.

What varies: when it happens (24 to 224 drawings in our runs). What does not: eventually all five take their left forks together, and nobody eats again. Python behaves the same with `threading.Lock` forks. Book: Section 14.11.1.

Two Clerks and Two Machines

Listing 14.21 `DeadlockDemo.java`, `clerk`: take the first machine, wait until the other clerk holds one too, then ask for 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
            }
        }
    }
```

```
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
```

In Listing 14.21 Ani takes the washer first and Budi the dryer first; a `CountDownLatch` makes the deadlock certain. `findDeadlockedThreads()` reports the cycle; `jstack` does the same from outside. Book: Section 14.11.2.

A Cycle in the Wait-For Graph

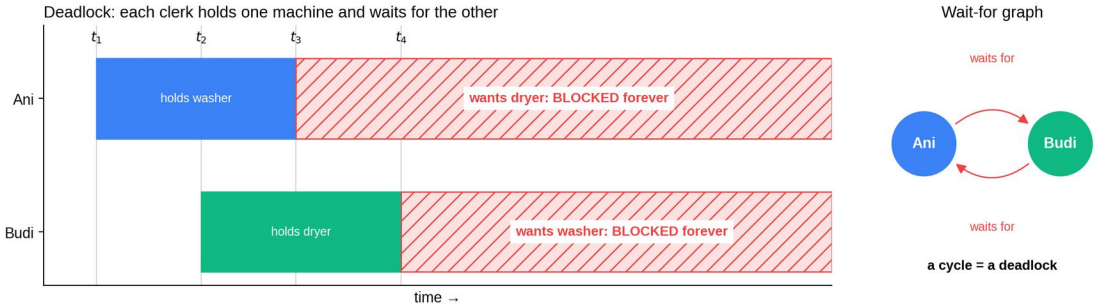


Figure 14.10 Timeline of the deadlock of Listing 14.21. At t_1 Ani takes the washer, at t_2 Budi takes the dryer, at t_3 and t_4 each asks for the other's machine and blocks forever. The wait-for graph on the right has a cycle, and a cycle in the wait-for graph is exactly a deadlock.

In Figure 14.10, an arrow runs from each blocked thread to the owner of the lock it wants. **A cycle in the wait-for graph is exactly a deadlock.**

Book: Section 14.11.2 and Figure 14.10.

The Four Coffman Conditions

> **DEFINITION.** Threads are **deadlocked** when each waits for a resource held by another in the set. Deadlock needs all four: (1) **mutual exclusion**; (2) **hold and wait**; (3) **no preemption**; (4) **circular wait**.

Break **any one** condition and deadlock becomes impossible. In the clerk example all four hold.

Python has no built-in detector: after a timeout both threads are alive and both locks held. Its `print` can merge two lines; a print lock (`say`) keeps them whole.

Book: Section 14.11.2.

Preventing Deadlock

Lock ordering: a thread holding ℓ_i may request ℓ_j only if

$$i < j \tag{14.5}$$

Along a chain of waits the numbers strictly increase, so no cycle can close.

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`

Table 14.3 names the Coffman condition each cure breaks; under Equation (14.5) only philosopher 4 takes the right fork first. Book: Section 14.12 and Table 14.3.

Ordering and an Arbitrator

Listing 14.22 `DeadlockFixes.java`, fix 1: 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]);
        });
```

Listing 14.23 `DeadlockFixes.java`, fix 3: a `Semaphore` lets at most four philosophers 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
```

Listing 14.22 breaks circular wait by order; Listing 14.23 by never letting the cycle close. The naive version ate 0 of 15 meals. Book: Section 14.12.

Try, and Back Off

Listing 14.24 `DeadlockFixes.java`, fix 2: `tryLock` with a timeout; on failure put the left fork down too, and wait a random time.

```
            while (true) {
                left.lock();
                try {
                    pause(10);
                    if (right.tryLock(5, TimeUnit.MILLISECONDS)) {
                        try {
                            pause(10);          // eat
                            return;
                        } finally {
                            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
            }
```

✔ **KEY INSIGHT.** Lock ordering is the workhorse: free at run time, no timeouts, checkable by reading. For objects of one class, lock the smaller unique number first.

The random pause in Listing 14.24 matters: two threads that fail together and retry together can fail together forever, which is livelock. Book: Section 14.12.

Livelock

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 `Livelock.java`: an owner that sees the other worker still active hands the resource over, up to `politeness` times.

```
                // 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);
```

In Listing 14.25, with unlimited politeness the resource passed back and forth for 100 ms and no job was done; the threads were never `BLOCKED`, which is why livelock is hard to spot. Book: Section 14.13.1.

Breaking the Symmetry

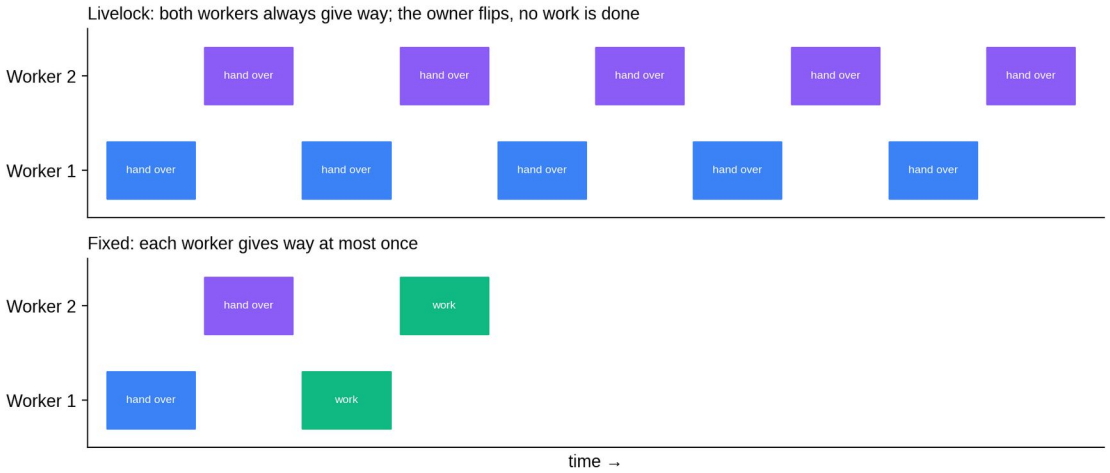


Figure 14.11 Livelock and its cure. Top: two always-polite workers hand the resource back and forth and never work. Bottom: each worker gives way at most once, so after two handovers both do their work.

In Figure 14.11: limit the politeness, give priorities, or back off for a *random* time. **Starvation**: a thread that never gets its turn; cure it with **fairness** (`new ReentrantLock(true)`, a FIFO queue).

Livelock is a failure of design, not of synchronization: the program has no data race at all. Book: Sections 14.13.1 and 14.13.2, and Figure 14.11.

The Concurrent Laundry

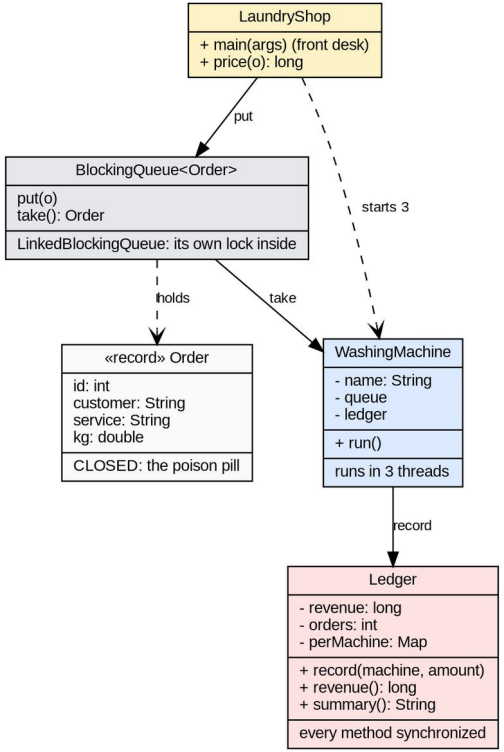


Figure 14.12 Class diagram of the concurrent laundry. The front desk (`main`) puts `Order` records into a `BlockingQueue`; three `WashingMachine` objects, each run by its own thread, take orders from it and record their prices in the single, fully synchronized `Ledger`.

In Figure 14.12: a **blocking queue** of orders, three **worker threads**, one **synchronized ledger**, and **poison pills** to switch the machines off.

Prices are those of Chapter 7: `WF` Rp7,000 and `WDI` Rp10,000 per kilogram, at least 3 kg, rounded up to the next half kilogram. Book: Section 14.14 and Figure 14.12.

The One Cashier

Listing 14.26 `LaundryShop.java`, `Ledger`: every method is `synchronized`, so 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;
        }
    }
```

Remove `synchronized` from `record` in Listing 14.26 and payments can be lost exactly as increments were; Exercise 14.3 asks you to try. Book: Section 14.14.

A Machine Is a Worker Thread

Listing 14.27 `LaundryShop.java`, `WashingMachine.run()`: take an order, wash it outside any lock, then pay.

```
        @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");
        }
```

*In Listing 14.27 the washing happens **outside the lock**: keep critical sections short. `Order` is an immutable record, and the pill `CLOSED` is recognized by identity. Book: Section 14.14.*

What Varied, What Never Did

```
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
```

Varied: which machine took which order, the work per machine (4, 4, 4 or 3, 5, 4), the switch-off order, the time. **Never:** 12 orders, each once, Rp533,500, equal to the one-thread total.

With m machines and washing times t_i, no schedule beats

$$T_{\text{wall}} \geq \max\left(\frac{1}{m}\sum_{i=1}^{k} t_i, \max_i t_i\right) \tag{14.6}$$

Equation (14.6) gives about 415 ms; "take the next order when free" traces to 498 ms, measured about 530 ms: "three machines" never quite means "three times as fast", in the spirit of Equation (14.1). Book: Section 14.14.

The Laundry in Python

Listing 14.28 `laundry_shop.py`, `WashingMachine.run()`: `get()` waits for an order; `None` is the pill; printing goes through `say()`.

```
    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")
```

Washing is waiting, so the GIL is released during every wash and the three machines in Listing 14.28 overlap as well as Java's: about 500 ms for 1,246 ms of washing. Book: Section 14.14.

Profitina in Practice

A request is a customer at the front desk: while one waits for the database, others are served (FastAPI on Uvicorn).

The scheduler is a producer, the scanners are consumers, separated by a queue, in the shape of Listing 14.27.

The local language model is the single cashier: requests wait their turn; with no model available, the application says so.

Counters that must not lose updates (login rate limits, PROFIT coin balances) live in PostgreSQL, whose transactions and row locks play the role of `synchronized`.

The software measures and explains; the user decides. Book: Section 14.15.

Common Mistakes

1. `run()` **instead of** `start()`**. 2. Shared data without a lock. 3. Guarding writes but not reads. 4.** `volatile` **for** `++`**.**

5. Trusting the GIL. 6. `if` **instead of** `while` **around** `wait()`**. 7.** `wait`**/**`notify` **without holding the lock.**

8. Locks in different orders (Equation (14.5)). **9. Slow work inside a lock. 10. Swallowing** `InterruptedException`**.**

11. Waiting forever: use timeouts. **12. Printing from Python threads without a lock. 13. Trusting a test that passed** (Equation (14.2)).

Each mistake has its cure in the book next to the program that shows it. Book: Section 14.16.

Summary — Five Things

1. **Threads share the heap**; the scheduler picks the interleaving, differently each run. Call `start()`, wait with `join()`, stop cooperatively (`interrupt()`, an `Event`, a `volatile` flag).
2. **The GIL**: Python threads help waiting, not pure-Python computing; speedups obey Equation (14.1).
3. **Races** lose updates in read-modify-writes; cure them with `synchronized`, locks or atomics, which also give visibility.
4. **Waiting** belongs in a `while` loop around `wait()`, or better in a `BlockingQueue` / `queue.Queue` with a poison pill.
5. **Deadlock** needs four conditions; break one, usually by lock ordering. **Livelock** is activity without progress; **starvation** is cured by fairness.

Book: Section 14.17.

Next Time

Chapter 15 — Sockets, Networking, and the Team Project. Programs on different computers talking over TCP, a protocol designed on paper first, and HTTP.

A server gives each client its own thread: today's worker threads, queues and locks meet the network.

Homework. Exercises 14.1–14.11; start with 14.2 (counting interleavings with Equation (14.2)) and 14.8 (two wallets without deadlock).

Solutions are in Section 14.19; try each exercise first. Book: Sections 14.18 and 14.19.

Lecture 15 — Sockets, Networking, and the Team Project

Object-Oriented Programming (EF234302)

Chapter 15 of the book, in miniature, for the classroom.

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*.

Lecturer: Irfan Subakti — yifana@gmail.com

Programs that talk to other programs, possibly written by other people in another language: first the network, then the people. Book: Chapter 15, opening paragraph.

Agenda

1. **Networking from zero.** IP addresses, ports, layers, TCP and UDP, clients and servers.
2. **Sockets.** A line-based echo server and client in Java and Python; flushing and framing; when the other side is not there.
3. **Protocols on paper first.** The knock-knock protocol as a table, a diagram and an object; many clients at once with a thread pool.
4. **Profitina Laundry.** An order-status service in JSON over TCP, Java and Python talking to each other; HTTP; security and UU PDP; testing.
5. **The team project.** Topic, components and interfaces, cross-testing, Git, failure handling, documentation, presentation.

The pages follow the book in order; the exercises at the end of the chapter are the homework. Book: Sections 15.2 to 15.11.

Where We Left Off

Every program so far **lived alone**: one computer, the keyboard, a file, the screen. Chapter 11 gave our programs windows and Chapter 14 gave them threads, but none spoke to another program.

Real software is sociable: the cashier's laptop asks a server whether ORD-0713 is ready; the courier's phone reports a delivery; the owner's dashboard asks for today's takings.

> ✔️ **KEY INSIGHT.** A network connection is just two streams, one each way, like the file streams of Chapter 5. What is hard: the other side is a separate program, it may disappear at any moment, and both sides must agree on the meaning of every byte.

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. Book: Section 15.1.

Hosts and IP Addresses

> **DEFINITION.** An **IP address** identifies a host's connection to a network. **IPv4**: 32 bits, written as four numbers 0–255, such as `192.168.1.10`. **IPv6**: 128 bits, eight hexadecimal groups, such as `2001:db8::10`.

Loopback: `127.0.0.1` and `::1`, "this very computer"; every example today uses it. **`localhost`** is a host name that DNS-style lookup turns into loopback.

Private addresses (`192.168.x.y`, `10.x.y.z`) live inside a home or office network; the laundry's server and the cashier's laptop have such addresses.

$2^{32} \approx 4.3$ *billion IPv4 addresses are far too few for a planet of phones; IPv6 is the long-term answer. Book: Section 15.2.1.*

Ports and the Four-Tuple

> **DEFINITION.** A **port** is a 16-bit number that the operating system uses to deliver network data to the right program. A server *listens* on a port; a client names it when it connects.

$$2^{16} = 65\,536, \text{ports } 0, 1, \ldots, 65\,535. \tag{15.1}$$

Well-known 0–1023 (80 HTTP, 443 HTTPS); registered 1024–49151 (we use 5050–5052); ephemeral 49152–65535. **Port 0** asks the system for a free one.

$$\text{connection} = (IP_c, p_c, IP_s, p_s) \tag{15.2}$$

By Equation (15.1) there are 65,536 ports per protocol; by Equation (15.2) one listening port serves thousands of clients, because each client gets its own ephemeral p_c. Book: Section 15.2.2.

How a Message Travels

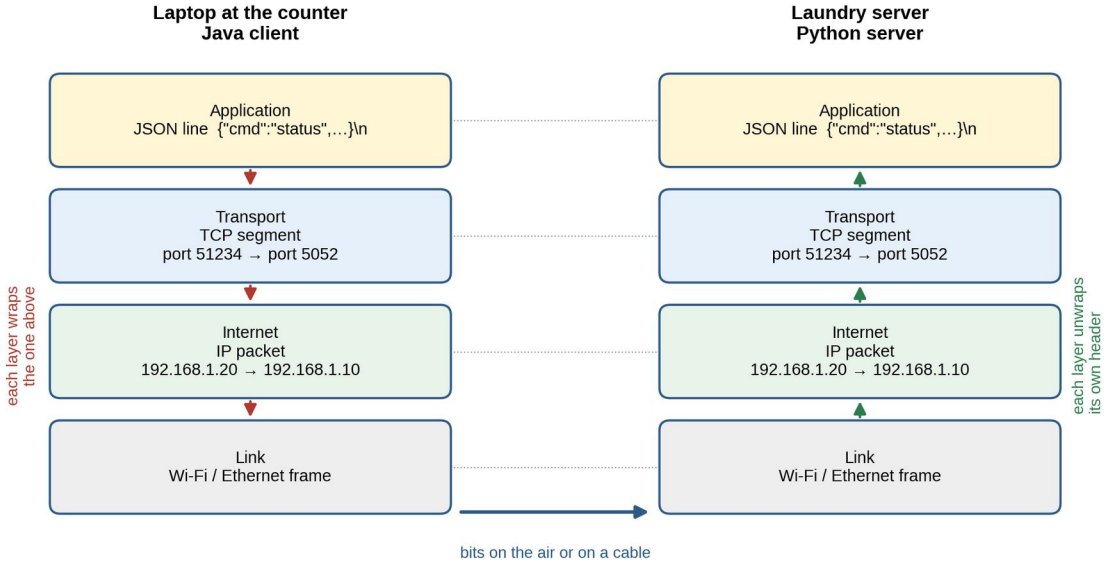


Figure 15.1 One request travelling through the four Internet layers. The client's layers wrap the JSON line in a TCP segment, an IP packet and a link-layer frame; the server's layers unwrap it in reverse. Each layer appears to talk directly to its twin on the other host.

In Figure 15.1: **link** (one hop), **internet** (IP, host to host, no promises), **transport** (TCP or UDP, program to program, ports), **application** (what the bytes mean). Each layer adds its own header on the way down.

Each layer seems to talk directly to its twin on the other host, the most useful illusion in networking. Book: Section 15.2.3 and Figure 15.1.

TCP and UDP

DEFINITION. TCP provides a *connection*: a reliable, ordered, two-way *stream of bytes*. **UDP** provides *datagrams*: independent messages, each whole or not at all, with no promise of delivery or order.

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

Table 15.1: TCP is a telephone call after a three-way handshake; UDP is a stack of postcards. A laundry cannot afford to lose "ORD-0713 was picked up", so we use TCP. Book: Section 15.2.4 and Table 15.1.

Datagrams to Ourselves

Listing 15.1 `UdpHello.java`: three datagrams sent to a receiver bound to port 0 on loopback, and received one whole datagram per call.

```
            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);
            }
```

```
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
```

In Listing 15.1, `getBytes(StandardCharsets.UTF_8)`: networks carry bytes, never characters. Three sends gave three receives of exactly 19, 16 and 18 bytes: UDP keeps boundaries. `setSoTimeout(2000)` stops a lost datagram from hanging `receive`. Book: Section 15.2.5.

The Same in Python

Listing 15.2 `udp_hello.py`: `SOCK_DGRAM` means UDP; addresses are `(host, port)` tuples.

```
    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')}")
```

Python's `socket` module in Listing 15.2 is a thin wrapper around Berkeley sockets (Unix, 1983); Java hides them behind a class per job. It prints exactly what the Java program printed. Book: Section 15.2.5.

Clients and Servers

DEFINITION. A **server** starts first, binds a socket to a known port and waits. A **client** knows the server's host and port and *initiates* the contact. The roles are about who starts, not the size of the computer.

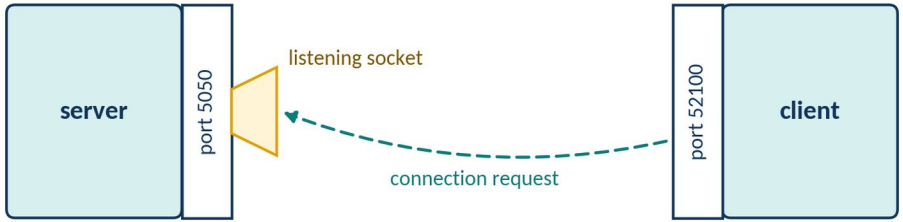


Figure 15.2 A client makes a connection request to the port on which the server is listening (here port 5050; the client's own port, such as 52100, is an ephemeral port chosen by its system).

In Figure 15.2 the client also binds to a local port of its own, usually assigned by the system. Book: Section 15.2.6 and Figure 15.2.

Listening and Connected Sockets

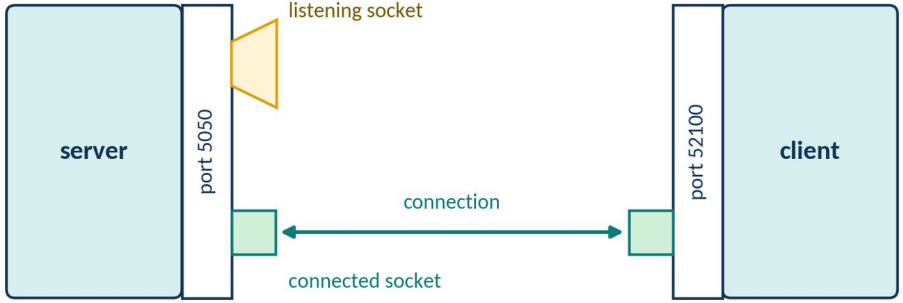


Figure 15.3 After acceptance the client and the server are connected through a new socket on the server's port 5050; the server's listening socket is still open for other clients.

In Figure 15.3, `accept` returns a **new** socket for the client; the **listening** socket stays open for the next one. It never carries data; each connected socket has its own four-tuple.

This is the most important idea for writing servers. Book: Section 15.2.6 and Figure 15.3.

Sockets in Java and Python

DEFINITION. A **socket** is one endpoint of a two-way communication link between two programs on a network, bound to a port so that the transport layer can identify the program data is for.

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`

Table 15.2 lists the calls in the order a program makes them. Book: Section 15.3 and Table 15.2.

The Life of One Connection

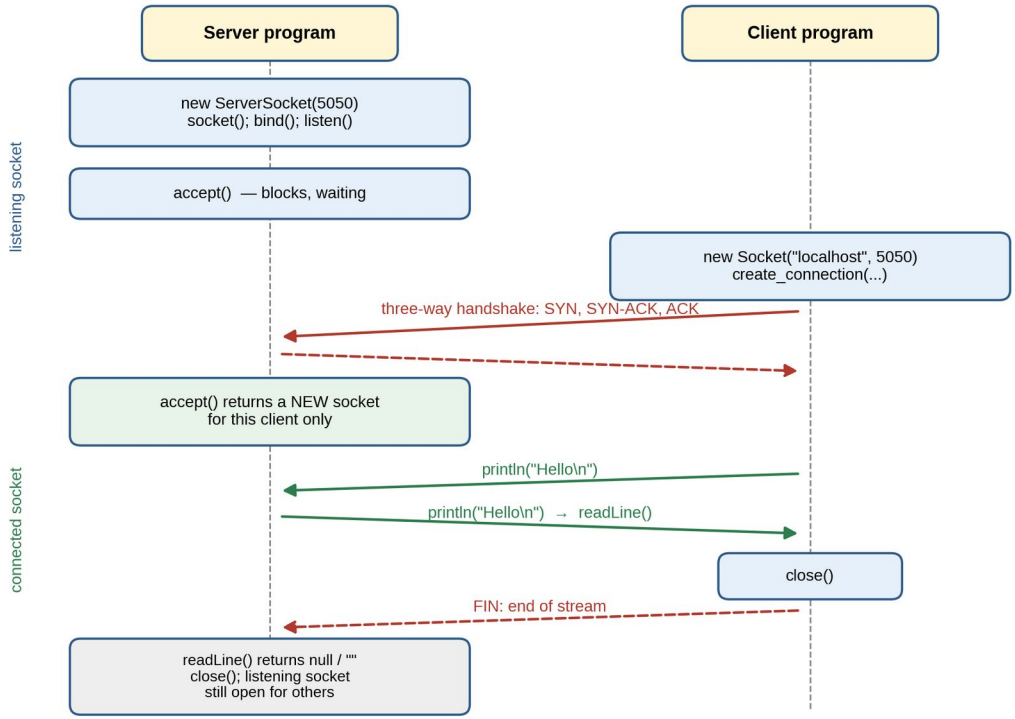


Figure 15.4 The life of one TCP connection. The server creates a listening socket and blocks in `accept()`; the client's connect triggers the three-way handshake; `accept()` returns a new connected socket; lines flow both ways; the client's `close()` sends FIN, which the server sees as the end of its input stream.

Wrap the streams (`BufferedReader`, `PrintWriter`; `makefile`). **Always name the encoding**, UTF-8. **Flush**: autoflush `PrintWriter`, `writer.flush()` in Python.

Forgetting to flush is the commonest reason a first network program "hangs". Figure 15.4 ends with the client's `close()`, which the server sees as end of input. Book: Section 15.3 and Figure 15.4.

An Echo Server in Java

Listing 15.3 `EchoServer.java`: listen on loopback, accept one client, and send every line back.

```
        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");
            }
```

In Listing 15.3, `accept()` blocks until a client arrives; `readLine()` returns `null` when the client closes; the backlog, 50, is how many connection requests may queue. Book: Section 15.4.1.

The Echo Server in Python

Listing 15.4 `echo_server.py`: `bind`, `listen`, `accept`, and a `for` loop over the client's lines.

```
        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")
```

In Listing 15.4, `SO_REUSEADDR` lets a restarted server bind at once despite TIME_WAIT (Java's `ServerSocket` sets it by default); `flush=True` makes the "listening" line appear even through a pipe. Book: Section 15.4.1.

The Echo Client

Listing 15.5 `EchoClient.java`: send each typed line, print the echo, and turn the three failures into sentences.

```
            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?");
        } catch (IOException e) {
            System.out.println("I/O error: " + e.getMessage());
        }
```

Five steps of every client: 1. open a socket; 2. open streams; 3. talk according to the protocol; 4. close the streams; 5. close the socket. Only step 3 differs.

In Listing 15.5, try-with-resources does steps 4 and 5; `ConnectException` *means nobody listens. Book: Section 15.4.2.*

The Echo Client in Python

Listing 15.6 `echo_client.py`: `create_connection` does step 1; `gaierror` and `ConnectionRefusedError` report failure.

```
            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}")
```

`create_connection` in Listing 15.6 tries every address the name resolves to (`localhost` may be both `::1` and `127.0.0.1`). Book: Section 15.4.2.

Two Programs, Two Terminals

```
Terminal 1:  java EchoServer 5050
Terminal 2:  java EchoClient localhost 5050
```

```
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
```

```
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
```

The Java client works with the Python server and vice versa: the **protocol**, not the language, decides who can talk to whom.

Start the server first. With no server running, both clients print `Cannot connect to localhost port 5050: is the server running?`. *Book: Section 15.4.3.*

Framing, and the Missing Other Side

DEFINITION. Framing is the rule that finds where one message ends inside a byte stream: a **delimiter** (a newline), a **length prefix**, or a **fixed size**.

`Hello\n` and `bye\n` may arrive as one piece or as three. A missing newline or a missing flush is a **deadlock across two programs**.

Server down: `ConnectException` at once. Client vanishes: `null` or `""`, or nothing at all, so use a **timeout**. Server vanishes mid-way: end-of-stream or "connection reset".

⚠ **WARNING.** Every blocking network call needs an answer to "and what if the other side never answers?": a timeout plus an exception handler.

Book: Sections 15.4.4 and 15.4.5.

A Protocol as a Table

DEFINITION. A **protocol** is the complete set of rules for a conversation: the messages, their exact format, their order, their meaning, and what each side does with a message it did not expect.

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`

Every row of Table 15.3 says what happens when the client says something else; those rows are where a network program spends much of its real life. Book: Section 15.5 and Table 15.3.

A Protocol as a Diagram

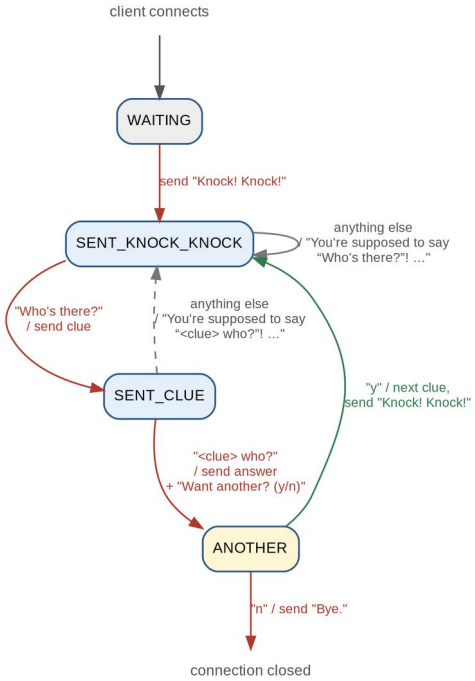


Figure 15.5 The knock-knock protocol as a state machine. Each arrow is labeled “what the client said / what the server answers”; grey arrows handle unexpected input; the green arrow starts the next joke.

> ✔️ **KEY INSIGHT.** Write the protocol three times before you run it: as a table, as a diagram (Figure 15.5), and as an object with no I/O. Each catches mistakes the others hide.

The table is better for checking that every case is covered, the diagram for seeing the shape; keep both in the repository. Book: Section 15.5 and Figure 15.5.

A Protocol as an Object

Listing 15.7 `KnockKnockProtocol.java`, `processInput`, first half: one `case` per state, as in the table.

```
    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!";
```

The protocol in Listing 15.7 ***knows nothing about sockets****: one object per conversation, a private* `enum` *state, and one method from the client's last line to the server's next line. Book: Section 15.5.*

Tested Without a Network

Listing 15.8 `KnockKnockProtocol.java`, `processInput`, second half: another joke, or `Bye.`.

```
            case ANOTHER:
                if (input.equalsIgnoreCase("y")) {
                    joke = (joke + 1) % CLUES.length;
                    state = State.SENT_KNOCK_KNOCK;
                    return "Knock! Knock!";
                }
                state = State.WAITING;
                return "Bye.";
            default:
                throw new IllegalStateException("unknown state " + state);
        }
    }
```

```
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.
```

`Iron?` is not `Iron who?`, so Listing 15.8 returns to `SENT_KNOCK_KNOCK`, exactly as the fifth row of Table 15.3 says. The `main` of the class plays a scripted conversation. Book: Section 15.5.

The Protocol in Python

Listing 15.9 `knock_knock_protocol.py`: `enum.Enum` states and a chain of `if` statements.

```
    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!"
```

Why a separate class? The protocol is tested with plain method calls; the socket code becomes a short loop that works for *any* line-based protocol; two teammates work in parallel, meeting at `processInput`.

Listing 15.9 prints exactly what the Java version printed. Book: Section 15.5.

Serving Many Clients at Once

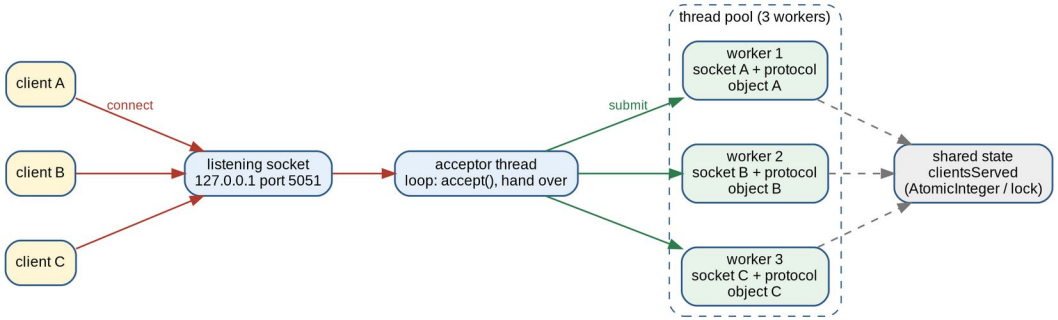


Figure 15.6 A multi-client server. The acceptor thread only accepts connections and hands each new socket to a worker of a fixed-size thread pool; each worker owns one socket and one protocol object; the only state the workers share is guarded by an atomic counter or a lock.

A thread per client costs stack memory: with n clients and stacks of size s,

$$M \approx n \cdot s \tag{15.3}$$

so a thousand idle clients at 1 MB reserve about 1 GB.

*Equation (15.3) is why servers use a **thread pool**, as in Figure 15.6; Java 21's virtual threads and Python's `asyncio` are the modern alternatives. Book: Sections 15.6 and 15.6.1, and Figure 15.6.*

The Front Door Only Accepts

Listing 15.10 `KnockKnockServer.java`, `acceptLoop`: accept, hand the socket to the pool, and accept the next one.

```
    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());
            }
        }
    }
```

In Listing 15.10, a slow client cannot block the door, because the acceptor never talks to a client itself. `start(port)` *returns the port actually bound, which matters for port 0. Book: Section 15.6.2.*

One Conversation per Worker

Listing 15.11 `KnockKnockServer.java`, `handle`: an idle timeout and a **new** protocol object for every client.

```
            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;
                }
            }
```

Shared fields use Chapter 14's tools: `AtomicInteger` counters, a concurrent set of open sockets, a `volatile` flag. **Share as little as possible, and protect what you must share.**

`stop()` closes the listening socket (the blocked `accept` throws), joins the acceptor, closes every open socket and shuts the pool down; Listing 15.11 counts each client in `finally`. Book: Section 15.6.2.

The Python Server

Listing 15.12 `knock_knock_server.py`, `_accept_loop`: a 0.2-second accept timeout lets the loop check the stop event.

```
    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
```

Closing a listening socket from another thread does not reliably wake a Python `accept()`, hence the timeout in Listing 15.12; `shutdown(SHUT_RDWR)` wakes workers blocked in `readline`. Book: Section 15.6.2.

Proving the Server Is Concurrent

Listing 15.13 `KnockKnockLoadTest.java`, `converse`: each greeted client waits at a `CyclicBarrier` until all three have been greeted.

```
        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());
            }
```

```
server stopped; clients served: 3
all clients were greeted before any finished: true
```

A sequential server would never greet the second client while the first waited; the barrier in Listing 15.13 would time out and the test would print `false`. The server runs on an ephemeral port inside the test. Book: Section 15.6.4.

The Order-Status Protocol

DEFINITION. JSON is a text format for structured data: objects `{"key": value}`, arrays, strings, numbers, `true`, `false`, `null`. One object per line is *JSON Lines*.

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

In Table 15.4 the client speaks first, every request gets exactly one response, and every response has `ok`. Python has `json`; the book's `Json.java` is a small recursive-descent parser. Book: Sections 15.7.1 and 15.7.2, and Table 15.4.

A Conversation with the Service

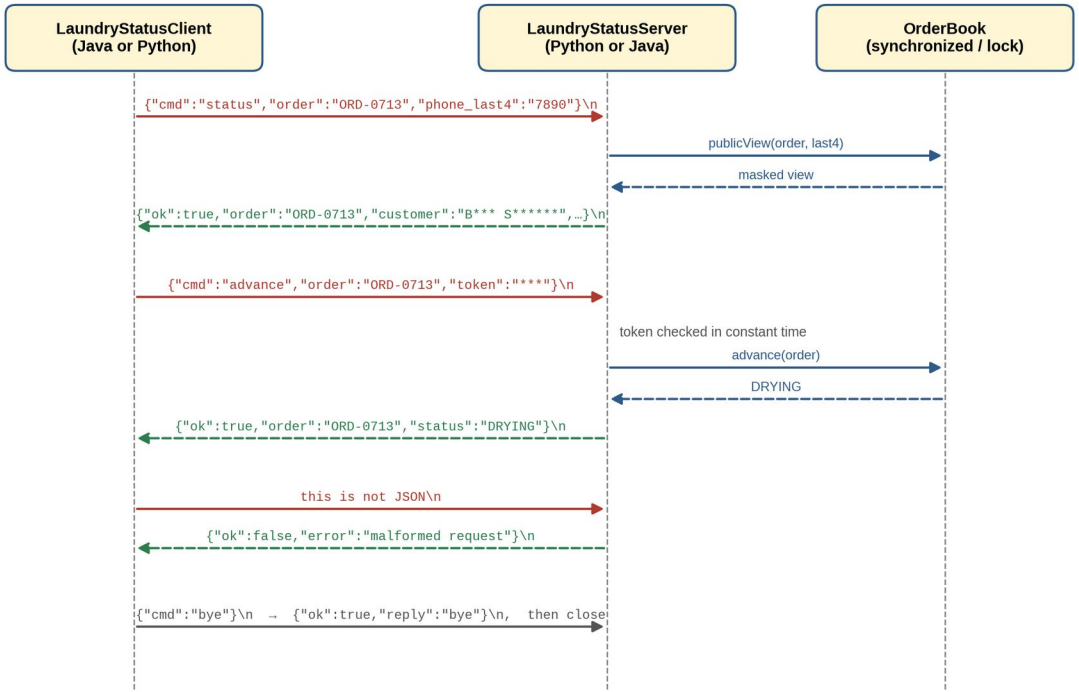


Figure 15.7 A conversation with the order-status service. Requests travel as JSON lines; the server consults the shared `OrderBook`, which is guarded by a lock; a malformed line receives an error, not silence; `bye` ends the conversation. The client may be Java and the server Python, or the other way round.

Four choices protect people in Figure 15.7: the last four phone digits are required; responses show initials only; a wrong order and a wrong phone get the *same* error; staff commands need a token from an environment variable.

Statuses follow Chapter 1's life cycle, `RECEIVED` to `PICKED_UP`. Book: Section 15.7.1 and Figure 15.7.

The Whole Protocol as One Function

Listing 15.14 `LaundryStatusServer.java`, `handle`, first half: `ping` and `status`, with every field checked before use.

```
    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");
```

`handle` *in Listing 15.14 maps a request map to a response map with no I/O, so it can be tested without a network.* `{"cmd":"status","order":42}` *must be refused, not crash a worker. Book: Section 15.7.3.*

Staff Commands Need a Token

Listing 15.15 `LaundryStatusServer.java`, `handle`, second half: `advance`, `bye` and `shutdown`.

```
        } 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");
        }
```

*In Listing 15.15, `isStaff` compares with `MessageDigest.isEqual`, in constant time. `OrderBook` is fully `synchronized`, and `mask` turns "Budi Santoso" into "B*** S******". Lines longer than 1024 characters are refused. Book: Section 15.7.3.*

The Server in Python

Listing 15.16 `laundry_status_server.py`, `handle`: the same rules, class for class.

```
    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")
```

In Listing 15.16, a `threading.Lock` replaces `synchronized`, `hmac.compare_digest` is the constant-time comparison, and `readline(MAX_LINE + 1)` limits a line directly. Book: Section 15.7.3.

A Client Object Hides the Wire

Listing 15.17 `LaundryStatusClient.java`: `status(...)` looks like an ordinary call.

```
public Map<String, Object> status(String order, String phoneLast4) throws IOException {
    return send(Json.object("cmd", "status", "order", order, "phone_last4", phoneLast4));
}
```

Listing 15.18 `LaundryStatusClient.java`: `send` writes one JSON line and reads one, the whole protocol from the client's side.

```
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);
}
```

*Listing 15.17 is a **proxy**: the class represents the service inside the client. Listing 15.18 throws when the server closes. The constructor connects with a 2-second limit and reads with a 5-second limit. Book: Section 15.7.4.*

The Scenario, Against the Table

```
> {"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"}
```

The right phone digits show the masked name; wrong digits and an unknown order get the same refusal; junk gets `malformed request` and the connection survives; the guessed token is refused, the real one moves `WASHING` to `DRYING`. Book: Section 15.7.4.

The Client Object in Python

Listing 15.19 `laundry_status_client.py`, `send` and `send_raw`; `__enter__` and `__exit__` make it work in a `with` statement.

```
    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()
```

Java talks to Python, Python talks to Java. `InteropDemo.java` starts the **Python** server as a separate process on port 0 and runs the Java scenario; `interop_demo.py` does the reverse.

```
> {"cmd":"shutdown","token":"***"}
< {"ok":true,"reply":"shutting down"}
Python server exited: true, exit code 0
```

With Listing 15.19 the two outputs are identical apart from the `server` field of `ping`: that is what a specification buys. The token for the run comes from `SecureRandom` and travels in the environment, never the command line. Book: Sections 15.7.4 and 15.7.5.

The Cost of a Conversation

One message of S bytes, latency L, bandwidth B:

$$T = L + \frac{S}{B} \tag{15.4}$$

k request–response pairs, server time t_s:

$$T(k) \approx k \cdot (RTT + t_s), RTT = 2L + \frac{S_{\text{req}} + S_{\text{resp}}}{B}. \tag{15.5}$$

EXAMPLE. 20 ms away over 10 Mbit/s: S/B for 180 bytes is ≈ 0.14 ms against $2L = 40$ ms. By Equation (15.5), ten requests cost about 400 ms; one request for ten orders about 40 ms.

In Equation (15.4) the latency dominates small messages: prefer a few rich messages to many small ones. Book: Section 15.7.6.

HTTP: The Protocol of the Web

Text lines on TCP, standardized: a request line (`GET /orders/ORD-0714 HTTP/1.1`), headers, an empty line, a body; the response starts with a status line (`HTTP/1.1 200 OK`).

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

In Table 15.5, 2xx means success, 4xx a problem with the request, 5xx a problem in the server. Framing: headers end at an empty line; `Content-Length` counts the body bytes. Book: Section 15.8.1 and Table 15.5.

HTTP in Java

Listing 15.20 `HttpStatusDemo.java`: `HttpClient` asks the in-process `HttpServer` three questions.

```
        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));
            System.out.println("GET " + path + " -> " + response.statusCode() + " "
                    + response.headers().firstValue("content-type").orElse("?"));
            System.out.println("    body: " + response.body());
```

```
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"}
```

In Listing 15.20 a 404 is a response, not an exception. `NO_PROXY` connects directly. The same program also writes a request by hand on a bare `Socket` and gets `HTTP/1.1 200 OK`: HTTP is only text over TCP. Book: Section 15.8.2.

HTTP in Python

Listing 15.21 `http_status_demo.py`, `do_GET`: the library parses the request and calls our method.

```
    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")
```

Listing 15.21 is the template method pattern of Chapter 7; `ThreadingHTTPServer` serves each request on a thread; `urllib` raises `HTTPError` for 4xx and 5xx. Profitina's back end speaks HTTP through FastAPI. Book: Section 15.8.2.

Seven Habits for Networked Code

1. **Listen only where you must** (`127.0.0.1` while developing). **2. Trust nothing** from the other side. **3. Limit everything**: line length, silence, workers.

4. Secrets out of code and logs. 5. Encrypt anything that leaves the machine: **TLS**. **6. Say little when refusing.**

7. Slow down guessers. Four digits, r attempts per second:

$$t_{\text{guess}} \approx \frac{10^4 + 1}{2 \cdot r}. \tag{15.6}$$

At $r = 1000$, Equation (15.6) gives about five seconds; five attempts per minute give about 17 hours: that is rate limiting. Book: Section 15.9.1.

What UU PDP Asks of Us

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

⚠️ **WARNING.** Never use real customers' data in a project, a test, a screenshot or a demonstration. Invent names and numbers.

Table 15.6: the log records `status -> ok`, not whose order; `publicView` returns initials, never the phone. Data protection is small decisions made while designing the protocol. Book: Section 15.9.2 and Table 15.6.

Testing Networked Programs

Test the protocol **without** the network; start a real server **on port 0** inside the test; a **fresh** server per test; test the **failures** on purpose.

Listing 15.22 `LaundryStatusCheck.java`, `check`: a fresh server with fresh data for every check, always stopped in `finally`.

```
    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();
        }
```

```
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
```

Listing 15.22 runs ten checks, including twenty clients at once and five staff clients who lose no update; the Python `unittest` suite does the same. Book: Section 15.10.

The Team Project: A Topic

The topic must *need* a client and a server: a game with chat, a small social network, a collaboration tool, an interactive portal, a startup idea.

Three tests. *Network*: does it need one? *Size*: can the core be shown after two thirds of the time? *Failure*: can you name a network failure that matters to the user?

Decide the topic, a working title and a one-paragraph pitch in the first meeting: who uses it, what for, and why it needs a network.

Book: Sections 15.11 and 15.11.1.

Five Components, Three Interfaces

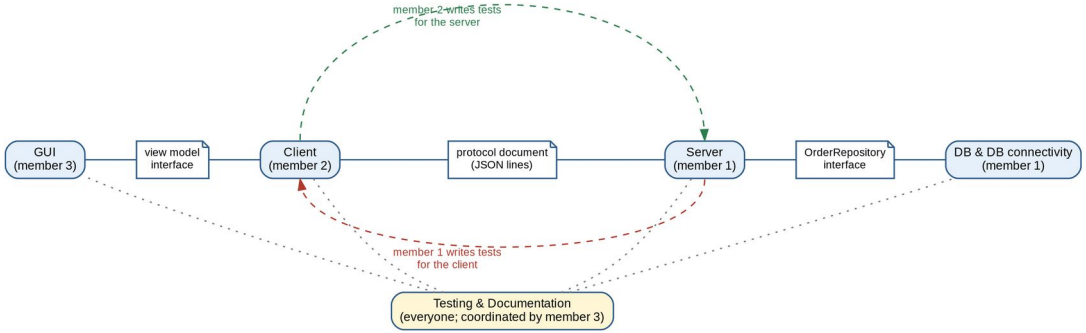


Figure 15.8 The five components of a team project, their owners, and the three interface documents between them. Dashed arrows show cross-testing: each owner writes tests for the component that uses theirs, or that theirs uses.

In Figure 15.8: server, client, database, GUI, testing and documentation, each with an owner. Between them: a **protocol document**, a **repository interface**, a **view model**.

Ownership means "answers for it and reviews every change", not "nobody else may touch it". Book: Section 15.11.2 and Figure 15.8.

Specification, Design, Implementation

Specification: one-sentence features with what counts as working. **Design**: components, interfaces, UML, state diagrams. **Implementation**: code and tests against the design.

Week 1 interfaces together; **week 2** in parallel against **stubs**; **week 3** replace stubs, cross tests; **week 4** failures, documents, rehearsal.

Cross-testing: when A uses B, A's owner designs B's tests. Unit, stub-based, multithreaded, failure and distributed tests.

The person who writes a component tests what they intended; its user tests what they assumed; bugs live in the gap. Book: Sections 15.11.3 and 15.11.4.

Working Together with Git

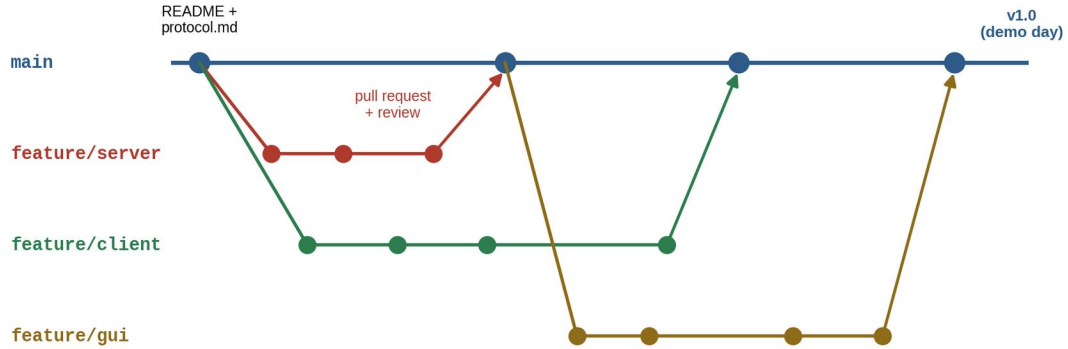


Figure 15.9 Feature branches. The team starts `main` with the README and the protocol document; each member works on a branch; every branch returns to `main` through a reviewed pull request; the demonstrated version is tagged.

In Figure 15.9: `main` always compiles and passes its tests; every change lives on a feature branch and returns through a reviewed **pull request**. Commit small and often; keep secrets out with `.gitignore`.

The first week's commands: `git clone`, `git switch -c`, `git add`, `git commit -m`, `git push`, `git pull`, `git merge main`. *Every repository starts with a README. Book: Sections 15.11.5 and 15.11.6, and Figure 15.9.*

Failure, Documents, Presentation

Server down: the client says so plainly, keeps the user's input, retries with growing waits. **Client down**: the server carries on and cleans up in `finally`.

Four documents: user guide, performance criteria in checkable numbers, installation guide, troubleshooting guide.

Present the problem in one sentence, demonstrate live on several computers, stop the server on purpose, show the tests, let every member speak, rehearse twice.

Nothing convinces like a failure handled well. Book: Sections 15.11.7 to 15.11.9.

A Sample Brief: Laundry Live Board

Staff register orders and move them through the stages; every screen updates within a second; customers check one order with its number and phone digits. Ayu: server and database; Bagus: client and protocol; Citra: GUI, tests and documents.

Table 15.7 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

Table 15.7: one deliverable per week, each checked by someone other than the person who produced it. A checklist for the week before the presentation closes the section. Book: Sections 15.11.10 and 15.11.11, and Table 15.8.

Profitina in Practice

app.profitina.com is a Python server (FastAPI on Uvicorn); every browser, phone and scheduled job is a client speaking HTTP and JSON.

`GET /api/me/dashboard` returns everything in one answer instead of a dozen requests: Equation (15.5) applied to a product.

The community feed uses **Server-Sent Events**; a Cloudflare Tunnel provides TLS; a release counts only when `/api/health` answers.

UU PDP turned into features: Argon2id hashing, consent, a login rate limit (Equation (15.6)), data export and purge. The application measures and advises; the user decides.

Book: Section 15.12.

Common Mistakes

1. Forgetting to flush. 2. Forgetting the newline. 3. Assuming one `recv` is one message. 4. Serving one client at a time.

5. Unprotected shared state between client threads. 6. Waiting forever. 7. "Address already in use": stop the old server, `SO_REUSEADDR`, port 0 in tests.

8. Mismatched encodings. 9. Binding to all addresses while developing. 10. Secrets in code or logs.

11. Personal data in errors and logs. 12. Coding before agreeing on the protocol.

Book: Section 15.13.

Summary — Five Things

1. **Addresses and ports.** A connection is its four-tuple (Equation (15.2)); TCP is a reliable byte stream, UDP independent datagrams.
2. **Sockets.** A listening socket produces one connected socket per client; wrap streams, name UTF-8, flush, frame with newlines.
3. **Protocols on paper first**: a table, a diagram, and an object without I/O that is tested alone.
4. **Many clients**: an acceptor and a thread pool; share little and protect it. JSON lines and a proxy make Java and Python interoperate; round trips dominate cost.
5. **People.** Written interfaces, cross-testing, Git with reviews, deliberate failure handling, UU PDP by design, and a rehearsed demonstration.

Book: Section 15.14.

Next Time

Chapter 16 — Evaluation and Discussion IV. Worked problems on Chapters 13 to 15 and on the whole book, each solved in full in Java and Python.

Iterators and immutability, threads and deadlock, sockets and protocols: every tool of the last three chapters used again, together.

Homework. Exercises 15.1–15.13; start with 15.1 (four-tuples with Equation (15.2)) and 15.3 (making the echo client freeze, then curing it with a timeout).

Solutions are in Section 15.16; try each exercise first. Book: Sections 15.15 and 15.16.

Lecture 16 — Evaluation and Discussion IV: Iterators, Concurrency, Networking, and the Whole Book

Object-Oriented Programming (EF234302)

Chapter 16 of the book, in miniature, for the classroom.

Book: Irfan Subakti, *Object-Oriented Programming with Java and Python*.

Lecturer: Irfan Subakti — yifana@gmail.com

No new language features today: fourteen worked problems, eight from one team project brief and six new ones, each solved in Java and in Python and checked by its own tests. Book: Chapter 16, opening paragraph.

Agenda

1. **A method** of ten steps for a whole-system project, and the project brief restated.
2. **Planning and protocol** (16.1–16.2): contracts, a joke store, the protocol as a state machine.
3. **Building and testing** (16.3–16.8): a multi-client server, cross tests with stubs, twenty clients at once, failures, a window that never freezes, measured documentation.
4. **Six new problems** (16.9–16.14): an iterator, a value object, a race, a wash line, a protocol of our own, and the capstone.
5. **Consolidation:** the problems compared, the whole book in one table, where to go next, common mistakes, summary and a closing word.

Each problem has the same shape: the problem, in our own words, design, the Java and Python solutions, line by line, a tempting wrong solution, and check yourself. Book: Section 16.1.

Where We Stand

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

Table 16.1 names the tools of Chapters 13 to 15; naming the tool is half of choosing it. Most programs today print the same lines on every run; the few that measure times or lost updates say so. Book: Section 16.1 and Table 16.1.

A Method for a Whole System

DEFINITION. A *contract* is a written promise at a boundary between two parts of a system: what one side may ask, what the other promises to answer, and what happens when something goes wrong. An `interface`, a network protocol and a file format are contracts.

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

Table 16.2: small goal, user stories, parts with owners, contracts first, the protocol as a state machine, cross tests with stubs, a walking skeleton, concurrency and failures on purpose, measure then document, write the report as you go. Book: Section 16.2 and Table 16.2.

The Project Brief

Build a client–server application with a GUI, talking through sockets, in a team of at most three: topic, division of work, specifications and interfaces, protocols and flow charts, cross tests, multithreaded tests, server-down and client-down handling, documentation.

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

*Our topic is the **Knock-Knock Club**: small enough to print in full, with every part the brief asks for. Table 16.3 maps each demand to one problem. Book: Section 16.3 and Table 16.3.*

Problem 16.1 — The First Contract

> **PROBLEM.** Choose the topic, divide the work, write the specification of each part, and document the interface between storage and the rest so precisely that another member can implement it; show with a test that two implementations keep the same promise.

Goal: a user connects, hears a joke, answers in the right places, and may ask for another. *Not a goal:* accounts, scores, user-written jokes.

The storage contract: at least one joke, numbered **from 0** to `size() - 1`; any other index throws `IndexOutOfBoundsException`. Writing the numbering down looks pedantic. It is the whole point.

Book: Section 16.4.1.

Five Parts and Their Contracts

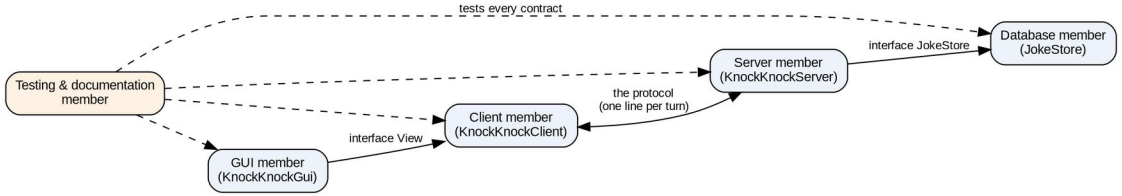


Figure 16.1 The five parts of the Knock-Knock Club and the contracts between them. Solid arrows are “uses”; dashed arrows show that the testing member tests every contract.

In Figure 16.1: database (member 1), server and client (member 2), GUI and testing with documentation (member 3). Contracts: `JokeStore`, the knock-knock protocol, `converse`, `View`.

Ownership means “I make it work and answer questions about it”, not “nobody else may touch it”. Book: Section 16.4.1 and Figure 16.1.

The Contract as an Interface

Listing 16.1 `JokeStore.java`: the storage contract with the immutable `Joke` record; the Javadoc of `get` *is* the specification.

```
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);
```

```
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
```

In Listing 16.1, `JokeStore.of` returns an in-memory stub from the first hour. The tempting wrong solution numbers from 1: the contract test catches it, `get(0)` throws and `get(4)` does not. Agreeing only "in words" is the deeper mistake. Book: Section 16.4.1.

Problem 16.2 — The Protocol, Drawn

> **PROBLEM.** Describe the protocol with a flow chart: “Knock, knock”; “Who’s there?”; a clue; “*clue* who?”; the punch line; “Want more?”. Implement the server’s side as a class that can be tested without any network.

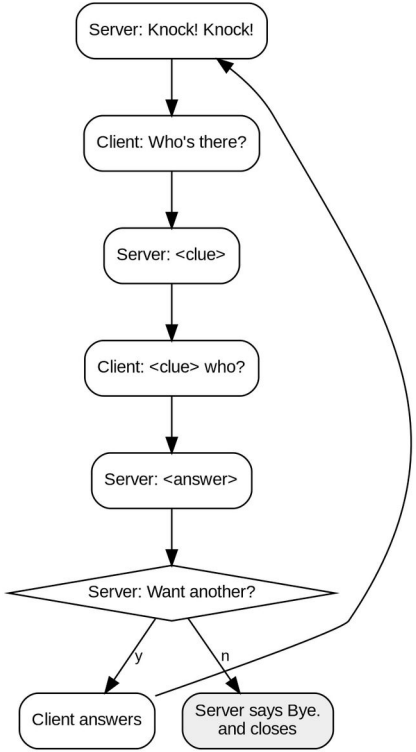


Figure 16.2 The knock-knock protocol as a flow chart: the conversation for people, with only the expected answers.

Figure 16.2 is a good picture of the happy path, but what if the client says "Who is it?", gives the wrong clue back, or answers "maybe"? Book: Section 16.4.2 and Figure 16.2.

The Protocol as a State Machine

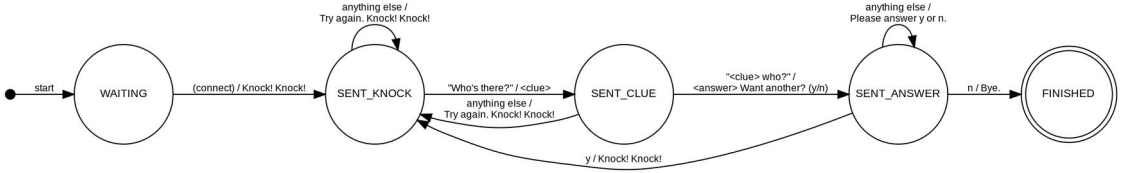


Figure 16.3 The same protocol as a state machine. Every state has an arrow for every possible input, including “anything else”.

DEFINITION. A *finite state machine* has states, a start state and a transition function

$$\delta(s, x) = (y, s') \tag{16.1}$$

from the state s and the client’s line x to the reply y and the next state s'.

Equation (16.1) is exactly the signature of `processInput`. In Figure 16.3 every state has an arrow for “anything else”. Framing: one line per turn, so punch line and “Want another? (y/n)” travel together. Book: Section 16.4.2 and Figure 16.3.

The Transition Table

Table 16.4 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`

Table 16.4 is the specification the implementation must match line by line; every state has a row for "anything else", which the flow chart does not show. Book: Section 16.4.2 and Table 16.5.

The Transitions in Code

Listing 16.2 `KnockKnockProtocol.java`, `processInput`, first half: `WAITING`, `SENT_KNOCK` and `SENT_CLUE`.

```
    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;
                }
                state = State.SENT_KNOCK;
                return "You're supposed to say \"" + joke.clue() + " who?\"! Try again. " + KNOCK;
```

In Listing 16.2, the two fields `state` and `current` are the whole memory of the conversation. `"Who's there?".equalsIgnoreCase(input)` puts the constant first, so a `null` input cannot throw. Book: Section 16.4.2.

Another Joke, or Bye

Listing 16.3 `KnockKnockProtocol.java`, `processInput`, second half: `(current + 1) % jokes.size()` wraps round the jokes.

```
            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");
        }
    }
```

```
PASS finished after n
PASS second joke told
PASS wrong greeting
PASS wrong clue
PASS not finished
PASS stateless reply is fooled
```

The `default` branch of Listing 16.3 is reached only after `Bye.`: calling the protocol then is a programming error. Python uses an `Enum` and a chain of `if` statements. Book: Section 16.4.2.

A Server Without Memory

Listing 16.4 `KnockKnockProtocol.java`, the tempting wrong `statelessReply`: it looks only at the last line.

```
    /** 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;
    }
```

Listing 16.4 answers “Lettuce in, it’s cold out here!” to “Cabbage who?”, and cannot tell a second joke. Whenever the right reply depends on what was said before, the object needs **state**.

> **CHECK YOURSELF.** Add the rule: in `SENT_KNOCK`, `quit` ends at once with `Bye.`. *Answer:* the row (`SENT_KNOCK`, `quit`, `Bye.`, `FINISHED`), and three lines inside `case SENT_KNOCK`; the client needs no change.

Book: Section 16.4.2.

Problem 16.3 — Many Clients at Once

PROBLEM. Build the server and the client. The server must serve several clients at the same time; show that a second client is served while a first is still in the middle of its conversation.

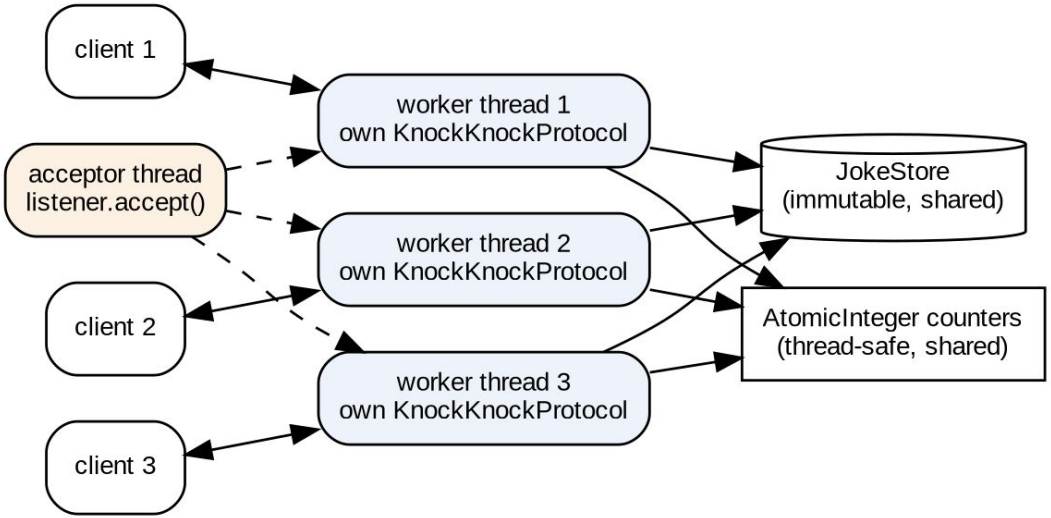


Figure 16.4 The threads of the knock-knock server. Each client has its own worker thread and its own protocol object; only immutable or thread-safe objects are shared.

In Figure 16.4 each client has its own worker thread and its own protocol object; the joke store (immutable) and the counters (`AtomicInteger`) are the only shared objects. Book: Section 16.5.1 and Figure 16.4.

Accept, and Hand Over

Listing 16.5 `KnockKnockServer.java`, `acceptLoop`: one thread per conversation, and the 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
        }
    }
```

The tempting wrong solution, `serve(client)` inside the loop, works perfectly with one client. With two, the second is connected but nobody talks to it until the first conversation ends.

In Listing 16.5, a user who goes to make coffee in the middle of a joke would block the whole club. Book: Section 16.5.1.

One Conversation, Testable with Strings

Listing 16.6 `KnockKnockServer.java`, `converse`: the server speaks first; `null` from `readLine()` means the client left.

```
    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;
    }
```

```
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.
```

`converse` in Listing 16.6 takes a reader and a writer, not a socket, so Problem 16.4 can test it with strings. Check yourself: ten thousand clients with `newCachedThreadPool`? Use `newFixedThreadPool(200)`, or virtual threads. Book: Section 16.5.1.

Problem 16.4 — Each Side Tests the Other

PROBLEM. Member A uses member B's functions, so A designs test cases for B, and B for A. Apply this to the client and server, with stubs, so that the tests need no running network.

DEFINITION. A *stub* is a minimal stand-in for a component the code under test depends on: fixed answers, no logic of its own.

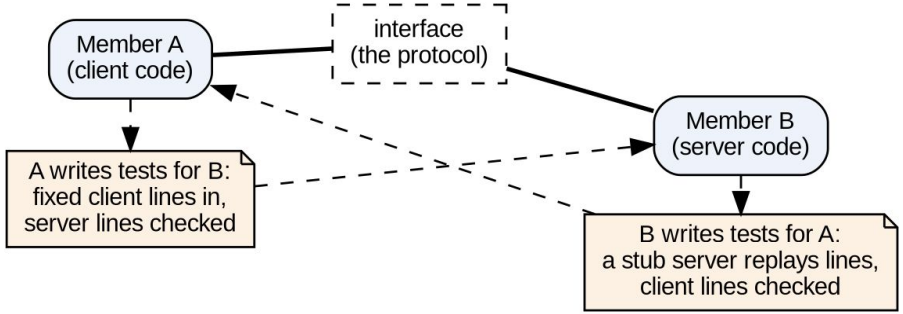


Figure 16.5 Cross testing through an interface. The client member writes tests for the server code; the server member writes tests for the client code. Each uses a stub of its own side.

In Figure 16.5, the person who uses a contract knows best what it must do. Book: Section 16.5.2 and Figure 16.5.

A Stub That Replays Lines

Listing 16.7 `ProtocolContractTest.java`: `replay` is the stub; `serverSays` is the client member's view of the server.

```
/** 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;
}
```

Listing 16.8 `ProtocolContractTest.java`, the tempting wrong `weakTest`: it only checks that the client said *something*.

```
/** 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();
}
```

Listing 16.8 passes a mutant that says "Who is there?"; the strong test compares exact lines and fails it: mutation testing. With Listing 16.7, `serverSays("Who's there?", "Cabbage who?")` ends with `(client left)`. Book: Section 16.5.2.

Problem 16.5 — Twenty Clients Together

> **PROBLEM.** Start many clients at the same moment against one running server; check that every client receives exactly the right conversation and that the server counted them all. Show a server that passes single-client tests but fails this one.

Oracle: the expected transcript is computed a second, independent way, without a network. **Start gate**: a `CountDownLatch(1)` releases all clients together. **Limits**: 30 turns, a time limit, and a two-second deadline for the count.

Book: Section 16.5.3.

The Start Gate

Listing 16.9 `ConcurrentClientsTest.java`, `correctConversations`: twenty tasks wait at the gate, start together, and compare with the oracle.

```
    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;
```

In Listing 16.9, `startGate.countDown()` releases everybody within microseconds; each `Future<Boolean>` holds one client's verdict. Polling with a deadline is how to test something that becomes true eventually. Book: Section 16.5.3.

One Protocol Object for Everyone

Listing 16.10 `ConcurrentClientsTest.java`, the tempting wrong server: one protocol object shared by every connection.

```
    /** 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());
```

```
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
```

Per-connection state belongs to the connection. What is shared must be immutable (the store) or protected (the counters).

Check yourself: would `synchronized processInput` *rescue Listing 16.10? No: calls no longer overlap, but A's knock followed by B's greeting still breaks both conversations. Book: Section 16.5.3.*

Problem 16.6 — When a Side Goes Down

PROBLEM. 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, and show that the server keeps serving others.

Table 16.5 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

*Table 16.5: every blocking call gets a **time limit**; **end of stream is a normal result**, `null` in Java and `""` in Python. Book: Section 16.6.1 and Table 16.6.*

Every Failure Becomes a Sentence

Listing 16.11 `FailureDetection.java`, `safeTalk`: connect with a limit, read with a limit, and catch from specific to general.

```
    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();
```

In Listing 16.11, `ConnectException` and `SocketTimeoutException` are subclasses of `IOException`; a general clause first would make them unreachable, a compile-time error. A transcript that does not end with `S: Bye.` means the server hung up early. Book: Section 16.6.1.

A Crash, or a Sentence

```
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]
```

Listing 16.12 `FailureDetection.java`, the tempting wrong `naiveTalk`: `line.equals(...)` on a `null` line.

```
    /** 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();
```

Listing 16.12 throws `NullPointerException`; the Python version loops on `""` until a `BrokenPipeError`. Check yourself: an unplugged cable sends nothing; give the server a read timeout, or a heartbeat. Book: Section 16.6.1.

Problem 16.7 — A Window That Never Freezes

> **PROBLEM.** Give the client a GUI that shows the conversation, sends each reply and shows the connection status. It must stay responsive while the network waits, and its logic must be testable without opening a window.

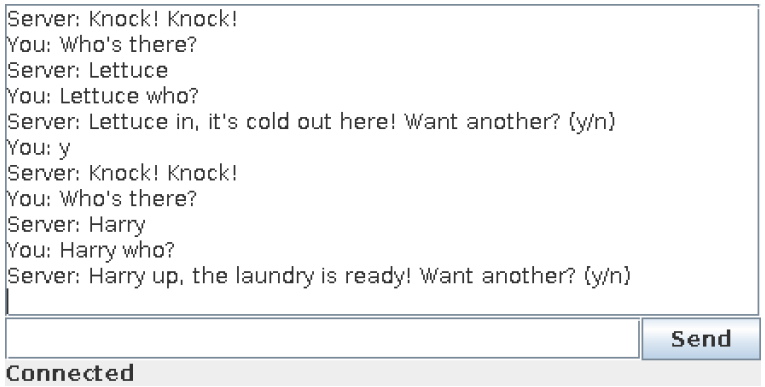


Figure 16.6 The Knock-Knock Club window (Swing) during a conversation. The status line reads “Connected”; the input field is enabled because the server is waiting for an answer.

Figure 16.6: the thread that draws the window must never wait for anything. MVC gives the structure: a `View` that only displays, a controller that does the work on its own `network` thread. Book: Section 16.6.2 and Figure 16.6.

The Network Thread Posts to the UI

Listing 16.13 `KnockKnockGui.java`, `connect`: the conversation runs on a daemon thread called `network`.

```
Thread connect(int port) {
    Thread network = new Thread(() -> talk(port), "network");
    network.setDaemon(true);
    network.start();
    return network;
}
```

Listing 16.14 `KnockKnockGui.java`, `talk`: every view change goes through `ui.execute`; the wait for the user happens on the network thread.

```
                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);
```

In Listing 16.13 `connect` returns at once. In Listing 16.14, `ui` is `SwingUtilities::invokeLater` in the window and `Runnable::run` in the test; Tkinter drains a `queue.Queue` every 50 ms. Book: Section 16.6.2.

Tested Without a Window

```
[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
```

The tempting wrong solution puts the socket code in the **Send** button's listener: while it waits, nothing repaints. The measurement shows the "repaint" blocked for 500 ms.

Check yourself: computing `String last` *before* `ui.execute` *hands the other thread only an immutable value, the same reasoning that made the joke store immutable. Book: Section 16.6.2.*

Problem 16.8 — Measured Promises

A **performance criterion** has a number and a condition. Throughput and Little's law:

$$X = \frac{n}{t} \tag{16.2}$$

$$L = X \cdot W \tag{16.3}$$

```
 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
```

Equation (16.2) counts conversations per second; Equation (16.3): 900/s at about 5 ms means about 4.5 in progress. Check yourself: 1000/s with 20 in progress gives $W = 0.02$ *s. The wrong criterion is one lucky run written as a fact. Book: Section 16.6.3.*

Documentation and the Report

Table 16.6 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 16.4
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 16.7
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)

The report: introduction, specification, design, implementation, testing, evaluation, conclusion, and **contributions**: each member's parts and an agreed percentage.

Table 16.6 lists the documents; the troubleshooting table on the next page completes them. Book: Section 16.6.3 and Table 16.7.

Troubleshooting in the Users' Words

Table 16.7 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

Table 16.7 lists symptoms as users see them, with the cause and the cure; the entries come from the failure scenarios of Problem 16.6. Book: Section 16.6.3 and Table 16.8.

Problem 16.9 — A Schedule with Its Own Iterator

> **PROBLEM.** Write `PickupSchedule`, an array always sorted by hour (stable for equal hours), usable in a for-each loop, with a *lazy* view of the pickups between two hours and an iterator that *fails fast* if the schedule changes during a loop.

Design. Fields `slots`, `size` and `modCount`; the iterator is an inner class `Cursor` with its own `next` and a copy of `modCount`; adding shifts later pickups right, as in insertion sort.

Book: Section 16.7.1.

A New Cursor for Every Loop

Listing 16.15 `PickupSchedule.java`, `Cursor`: an inner class with its own position and a fail-fast check.

```
    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++];
        }
    }
```

`iterator()` returns `new Cursor()` every time, which is what makes nested loops count 25 pairs. Listing 16.15 throws `ConcurrentModificationException` when the loop removes A4. Book: Section 16.7.1.

The Collection Is Not the Iterator

Listing 16.16 `PickupSchedule.java`, the tempting wrong `SelfIteratingSchedule`: `iterator()` returns `this`.

```
    /** 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"
        }
```

```
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
```

Nested loops over n elements visit

$$P_{\text{own}} = n^2, P_{\text{shared}} = n - 1 \tag{16.4}$$

pairs: 25 and 4 for $n=5$.

Equation (16.4) explains the output of Listing 16.16. Check yourself: removing A1, the second-to-last, ends the loop silently: fail-fast is only "best effort". Book: Section 16.7.1.

The Schedule in Python

Listing 16.17 `pickup_schedule.py`: `__iter__` returns a new `_Cursor`; `between` is a generator.

```
    def __iter__(self):
        return _Cursor(self)                    # a NEW position for every loop

    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
```

Listing 16.18 `pickup_schedule.py`, `_Cursor.__next__`: `StopIteration` ends the loop.

```
    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
```

A generator object from Listing 16.17 is an iterator: `len(list(view))` is 3 the first time and 0 the second; `s.between(8, 12)` makes a fresh one on each call. Listing 16.18 raises `RuntimeError` for a changed schedule. Book: Section 16.7.1.

Problem 16.10 — Money as a Value

PROBLEM. Write `Money`: whole units and a three-letter currency. It never changes; adding or pricing makes new objects; equal amounts are equal and work as the same hash key; different currencies are refused.

DEFINITION. The *hash contract*: if `a.equals(b)`, then `a.hashCode() == b.hashCode()`. A hash map looks only in the bucket of the key's hash code.

Design. A `final` class, `private final` fields, a private constructor behind a validating factory `of`, amounts in a `long`, never a `double`.

Book: Section 16.7.2.

equals and hashCode from the Same Fields

Listing 16.19 `Money.java`: `equals` compares both fields; `hashCode` is built from exactly the same fields.

```java
    @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
    }
```

```
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
```

In Listing 16.19 `a == b` is false (two objects) and `a.equals(b)` true (one value). The last two lines are the wrong variants: without `hashCode` a set holds two "equal" elements; a mutated key becomes a ghost. Book: Section 16.7.2.

Money in Python

Listing 16.20 `money.py`: a frozen dataclass writes `__eq__` and `__hash__`; `__post_init__` validates.

```python
@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}")
```

Python refuses the first mistake outright: `__eq__` without `__hash__` makes a class unhashable. `round` rounds halves to even; write the rounding rule in the specification.

Check yourself for Listing 16.20: IDR 5000 does not equal USD 5000; `rupiah(0)` is allowed; `rupiah(-1)` throws, so no invalid `Money` ever exists. Book: Section 16.7.2.

Problem 16.11 — A Measured Race

Four machines, four threads, one shared counter. Expected total and lost updates:

$$N = T \cdot k \tag{16.5}$$

$$L_{\text{lost}} = N - c\,. \tag{16.6}$$

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.

With $T = 4$ and $k = 1,000,000$, Equation (16.5) gives 4,000,000; Equation (16.6) measures the damage. Figure 16.7: both machines read 41, both write 42. Book: Section 16.8.1 and Figure 16.7.

Widening the Window on Purpose

Listing 16.21 `WashCounter.java`, `SlowUnsafeCounter`: the gap 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
        }
```

```
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
```

Listing 16.21 turns "fails once a month" into "fails every run": 50 of 200. The volatile unsafe counter lost 7,834 to 60,543 of four million; these numbers vary between runs. Book: Section 16.8.1.

Three Cures, and a Surprise

`synchronized` methods; a private lock object; `AtomicLong.incrementAndGet()`, a compare-and-swap. All three reach exactly 4,000,000.

In CPython the plain unsafe counter lost *nothing*: an implementation detail, not a promise. The slow counter proves the code wrong regardless; `with self._lock:` makes it exact.

> ⚠️ **WARNING.** The tempting conclusion, "`+=` is atomic in Python", is wrong. If two threads use one variable and one writes, protect it, whether or not a test has ever failed.

Check yourself: with 4 threads in lock step, each round adds 1 instead of 4, so 50 rounds give exactly 50. Book: Section 16.8.1.

Problem 16.12 — A Wash Line

> **PROBLEM.** The desk puts orders on a shelf of at most two; two machines take them. Write the shelf with `wait`/`notifyAll` (Python: `Condition`), then with a blocking queue; stop the machines cleanly; check every order is washed exactly once.

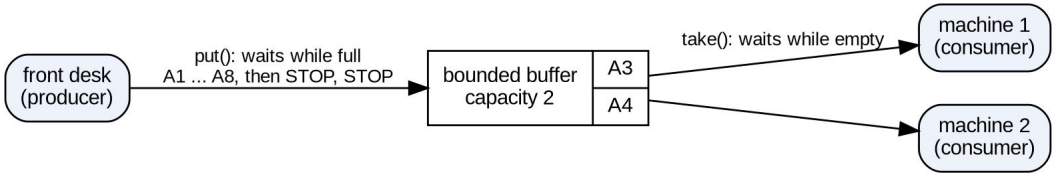


Figure 16.8 The wash line: one producer, a bounded buffer of capacity 2, and two consumers. Two `STOP` pills, one per machine, end the day.

In Figure 16.8, one `STOP` pill per machine ends the day. Book: Section 16.8.2 and Figure 16.8.

A Hand-Made Monitor

Listing 16.22 `WashLine.java`, `OrderBuffer`: `put` and `take` wait in `while` loops and wake everyone with `notifyAll`.

```
    @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
    }

    @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;
    }
```

In Listing 16.22, `wait()` releases the lock, sleeps, and re-acquires it; the `while` re-checks after spurious or stale wake-ups; `notify()` might wake the wrong kind of thread. In real programs use `ArrayBlockingQueue`. Book: Section 16.8.2.

if Instead of while

Listing 16.23 `WashLine.java`, the tempting wrong `IfBuffer.take`: the condition is checked once.

```java
        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 16.24 `wash_line.py`, `OrderBuffer.take`: the right way with `threading.Condition`.

```python
    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
```

```
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]
```

With Listing 16.23, both machines wake for one order; the second removes from an empty deque. Listing 16.24 waits in a `while`. Check yourself: one pill for two machines leaves one waiting forever. Book: Section 16.8.2.

Problem 16.13 — LAUNDRY/1

PROBLEM. Design `LAUNDRY/1`: the server greets `LAUNDRY/1 READY`; the client sends `SERVICES`, `PRICE <service> <kg>` or `QUIT`; every request gets exactly one line starting with `OK` or `ERR`. Test it with ten clients at once.

Table 16.8 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 ...`

In Table 16.8 each request stands alone, so the server needs no conversation state: the whole protocol is a pure function. Book: Section 16.9.1 and Table 16.9.

The Protocol as a Pure Function

Listing 16.25 `LaundryProtocol.java`, `reply`, the `PRICE` case: one `ERR` branch per row of the table.

```
            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];
                }
                if (kg <= 0 || kg > 50) {
                    return "ERR weight must be in (0, 50] kg";
                }
                return "OK " + Math.round(rate * kg);
            default:
```

Listing 16.25: $7000 \times 3.5 = 24,500$ *and* $14,000 \times 2 = 28,000$. *Most of the protocol can be tested with no sockets; the network test checks framing and concurrency. Book: Section 16.9.1.*

The Same Function in Python

Listing 16.26 `laundry_protocol.py`, `reply`: `split()` without an argument splits on runs of white space.

```
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}"
```

Listing 16.26 returns exactly the lines the Java version returns. Book: Section 16.9.1.

The Request That Never Left

```
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)
```

Listing 16.27 `LaundryProtocol.java`, the tempting wrong `NoFlushClient`: a `PrintWriter` without autoflush.

```
    /** 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));
        }
    }
```

Ten clients sum to $7000 \times 55 = 385,000$. *Listing 16.27 keeps the request in its buffer; only the timeout saves it from hanging. Check yourself:* `QUOTE WASH 3.5` *gives* `OK 26950`*, and only* `reply` *changes. Book: Section 16.9.1.*

Problem 16.14 — The Capstone

PROBLEM. A small but complete laundry: orders with a service and a weight, an identifier and a price, statuses from received to picked up, reports, a file that survives a restart, several counters at once through a socket API; classes, an interface, inheritance, collections, a checked exception, files, MVC, in both languages, with tests.

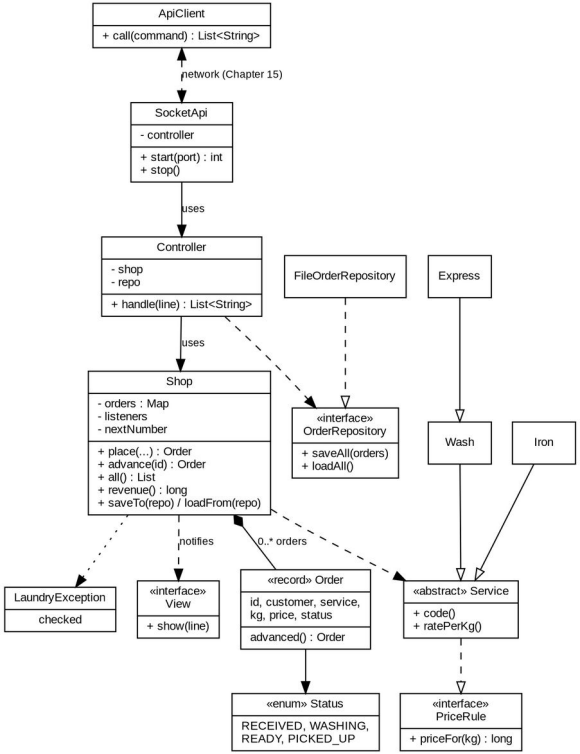


Figure 16.9 UML class diagram of the capstone. Hollow triangles mean “extends” (solid line) or “implements” (dashed line); the diamond means that the shop holds the orders.

In Figure 16.9: `SocketApi`, `Controller`, `Shop` (the model, with observers) and the smallest blocks. `LIST` has many reply lines, so every reply ends with a line holding only a dot, the SMTP rule. Book: Section 16.10.1 and Figure 16.9.

An Express Wash Is a Wash

Listing 16.28 `LaundryCore.java`, `Service`: the template method `priceFor` built from two abstract methods.

```
    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 16.29 `LaundryCore.java`, `Express extends Wash`: twice the washing rate, through `super`.

```
    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
        }
    }
```

Listing 16.28 is Chapter 7's template method; Listing 16.29 follows any change of the washing rate automatically. Check yourself: dry cleaning is one new class and one map entry; `PLACE Ayu DRYCLEAN 1.5` *gives* `OK L1 30000`*. Book: Section 16.10.1.*

One Place Where Orders Change

Listing 16.30 `LaundryCore.java`, `Shop.advance`: `synchronized`, and the immutable order is *replaced*, never modified.

```
    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;
    }
```

```
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
```

Without `synchronized` in Listing 16.30, two threads could read the same `nextNumber`: the lost update of Problem 16.11 in a new place. Five clients placed twenty orders with twenty different ids. Book: Section 16.10.1.

A Session at the Counter

```
> 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
```

Every refusal is an `ERR` line, never a dead connection thread. The tempting wrong client reads one line per reply; after `LIST` it is out of step for ever, and `REVENUE` "returns" order L2. Book: Section 16.10.1.

The Fourteen Problems Compared

Table 16.9 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

Read down the last column of Table 16.9: almost every wrong solution passes with one client, one thread or one line, and fails only under the condition the problem was about. Book: Section 16.11 and Table 16.11.

The Whole Book in One Table

Table 16.10 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

In Table 16.10 the questions change character halfway: from "how do I write this?" to "how do these parts work together, when several things happen at once?". Book: Section 16.12 and Table 16.12.

Where to Go Next

Design patterns by name: Iterator, Observer, MVC, Strategy, Template method, Factory method, Repository, Producer–consumer, State. Solutions to problems, not goals.

Build tools: Maven (`pom.xml`, `mvn test`) and Gradle; Python virtual environments, pip, `pyproject.toml`, pytest; Git and continuous integration from day one.

Frameworks call your code: Spring Boot and JavaFX; FastAPI, Django, Flask. Learning one is learning where it puts the pieces you already know.

A reading path: Bloch, Goetz et al., Ramalho, Gamma et al., Fowler's Refactoring, Brooks's The Mythical Man-Month. Book: Section 16.13.

Profitina in Practice

Contracts: every scanner signal behind one interface, checked by the same contract tests; well over a thousand tests gate every release.

Many clients: Server-Sent Events push posts to every open browser; `GET /api/me/dashboard` is one request, one complete reply.

Failure handled honestly: a failed post restores the text; a release counts only when `/api/health` answers; without a local model the application says so.

Personal data in one place, under UU PDP 27/2022. The product advises; its users decide.

Book: Section 16.14.

Common Mistakes

Contracts only in words; a protocol without an answer for every input; no framing rule; serving one client at a time; sharing per-connection state.

Tests that cannot fail; blocking calls without a limit; forgetting the end of the stream; network code on the UI thread; performance claims from one run.

`iterator()` returning `this`; `equals` without `hashCode`, mutable keys; "the test passed, so no race"; `if` around `wait`; forgetting to flush.

Each mistake points to the problem that shows it. Book: Section 16.15.

Summary — Five Things

1. **Contracts first**: interfaces and protocols written before the code, each with a test; a protocol is a state machine, Equation (16.1).
2. **Servers**: a thread and a state per connection; cross tests with stubs; many clients through a start gate against an oracle.
3. **Robustness**: time limits, end of stream as a normal result, a window that never waits, criteria measured with Equation (16.2) and Equation (16.3).
4. **Iterators and values**: a new iterator per loop (Equation (16.4)); immutable values with `hashCode` agreeing with `equals`.
5. **Threads and protocols**: races lose updates (Equation (16.6)); `while` around `wait`; pure-function protocols that flush; the capstone joins the whole book.

Book: Section 16.16.

A Closing Word

You have 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.

> ✔ **KEY INSIGHT.** Ask of every part of a system what it knows, what it does, whom it trusts and what it promises; write that promise down; test it before believing it. That habit belongs to no single language.

Build something you care about, build it with others, and then build the next thing a little better. Book: Section 16.16, “A closing word”.

What Comes Next

Exercises 16.1–16.12: `Countdown`, removing during a loop, splitting money, a race by hand, deadlock between two machines, `TOTAL`, heartbeats, an SQLite store, troubleshooting, a `READY` counter, test design, a Maven layout.

Start with 16.4 (the smallest possible result of two racing threads) and 16.7 (a heartbeat for the knock-knock protocol).

Then the team project: choose the topic in the first meeting, and write the contracts before the code.

Complete solutions, with Java and Python programs, are in Section 16.18. Book: Sections 16.17 and 16.18.