Object-Oriented Programming

First Edition

Irfan Subakti

JAVA TRACK

Profitina
profitina.com

TABLE OF CONTENTS

Preface

This book grew out of the Object-Oriented Programming course I teach in the Department of Informatics at Institut Teknologi Sepuluh Nopember (ITS) Surabaya, and it is organised the way the course itself is organised: 16 chapters, each one a session you could sit through, work through, and then use. This is the Java edition; a companion edition covers the same 16 chapters in the other language.

Before you write a single class, you need to see the problem object-oriented programming was invented to solve. This book builds that case with a small library system — Pustaka — whose procedural version starts breaking as it grows, and rebuilds it, concept by concept, first in Java and then in Python.

Object-Oriented Programming (OOP) exists as two parallel editions — one in Java, one in Python — that teach the exact same sixteen-stop roadmap, differing only in language idiom, so a reader who already knows one edition can use the other as a direct comparison of how the same idea looks in each language. The course opens by showing why a procedural library-management program (Pustaka) outgrows itself, then rebuilds it as classes with fields, constructors and encapsulation; adds arrays/lists and string-building; covers exception handling and file I/O; introduces interfaces and abstract classes; and moves into inheritance, polymorphism, generics and the collections framework, GUI programming with events and MVC, concurrency (threads, race conditions, synchronization), and networking with sockets. It closes by naming, in Design Patterns, what the Pustaka codebase has already been doing by then — and finishes with a capstone GUI-socket client-server project.

By the time you reach the last page, you should be able to:

- Recognize when and why a procedural program needs to become object-oriented, not just how to write class syntax.
- Design classes around fields, constructors, and encapsulation, in both Java and Python idiom.
- Use inheritance, polymorphism, interfaces/abstract classes, and generics to build flexible, reusable code.
- Handle exceptions and file I/O robustly, and build a graphical interface with an event-driven MVC structure.
- Write concurrent code safely — threads, race conditions, synchronization — and network code with sockets, clients and servers.
- Recognize common design patterns as names for structures your own code has already converged on.

Where this book needs a real system to point at, it points at Profitina — profitina.com and app.profitina.com — a platform I design, build and operate myself. That choice is practical, not promotional: I can show you the actual decision, the actual trade-off, and the actual consequence, because I was the one who had to live with it. None of those examples are retrofitted analogies; where they appear, they are the same problem, examined from a teaching angle.

Who this book is written for: students who have already taken a Fundamentals of Programming or Data Structure course and can write procedural code, and now need object-oriented design — in Java, Python, or both.

A word on method. Every code example in this book was compiled and run before it was written down, and the appendices carry the verification logs to prove it. Where a number appears, it came from a measurement rather than from memory. If you find an error, or a passage that could be clearer, I would genuinely like to hear about it — that is how the next edition gets better.

Surabaya, September 2026

Irfan Subakti
yifana@gmail.com

Profitina

profitina.com

CHAPTER 1

Introduction to OOP: From Procedures to Objects

Before you write a single class, you need a clear picture of the problem object-oriented programming was invented to solve — and why "just write more functions" eventually stops working.

Learning Objectives — after this chapter you will be able to:

(1) Explain, with a concrete example, why large procedural programs become hard to change safely. (2) Define an object as a bundle of data (fields) and behavior (methods), and explain encapsulation as "hiding the data behind the behavior." (3) Name the four pillars of OOP (encapsulation, inheritance, polymorphism, abstraction) and preview where each is taught in this course. (4) Install and verify a working Java Development Kit (JDK) and IDE. (5) Read and explain every line of a minimal Java class and its accompanying driver program, and compile and run it yourself.

One Toolchain, Three Operating Systems — Windows 11, macOS, Ubuntu

OS	One-time install (JDK 21 LTS)	Compile & run (identical everywhere)
Windows 11	winget install EclipseAdoptium.Temurin.21.JDK	javac Main.java then java Main
macOS	brew install --cask temurin@21 (or adoptium.net installer)	javac Main.java then java Main
Ubuntu/Linux	sudo apt install openjdk-21-jdk	javac Main.java then java Main

Every example in this book runs identically on Windows 11, macOS, and Ubuntu/Linux; commands differ only at install time. Pick your row once and follow along on your own laptop.

1.1 Welcome to Object-Oriented Programming

Welcome to Object-Oriented Programming (OOP). If you have already taken a Fundamental Programming or Data Structure course, you already know how to write functions, loops, arrays, and conditionals — the procedural toolkit. This course does not throw that toolkit away. Instead, it teaches you a different way of organizing the same tools: instead of scattering data across loose arrays and passing that data between free-floating functions, you will learn to bundle data and the functions that operate on it into a single unit called an object. That single idea — data and behavior, bundled together, hiding their own internal details — is the seed every other concept in this course grows from: encapsulation, inheritance, polymorphism, interfaces, generics, and eventually the design patterns used throughout real production software, including the very platform whose brand appears on this book's cover.

This course teaches every concept twice: once in Java (the classic, statically-typed, industry-standard OOP language this course has always used) and once in Python (a dynamically-typed language whose OOP features are more lightweight but conceptually identical). This is the Java edition. A companion Python edition exists with the exact same 16-lecture roadmap and the exact same worked examples, translated into idiomatic Python — so that whichever language your team, employer, or personal project eventually needs, you already understand the underlying ideas and only need to learn new syntax, not new concepts (Figure 1.1).

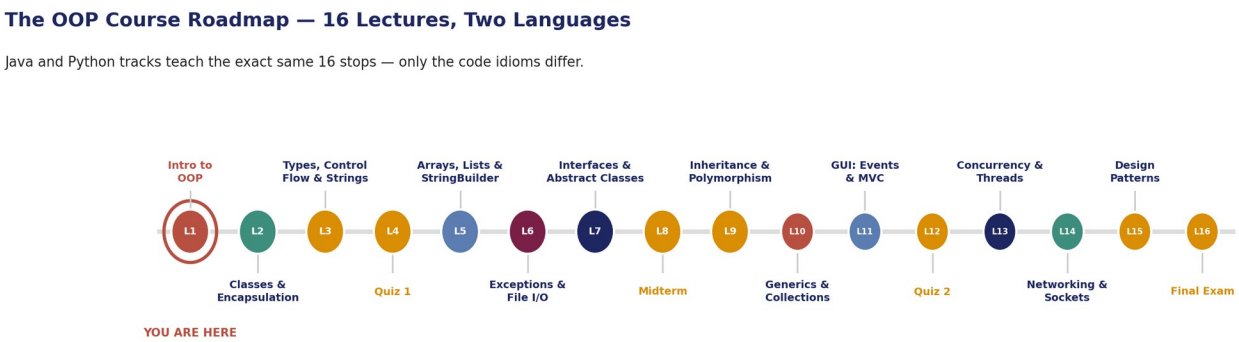


Figure 1.1 — The 16-lecture OOP roadmap. Both the Java and Python tracks visit the exact same 16 stops; only the code idioms differ. You are here: Lecture 1.

1.2 Why Procedural Programs Outgrow Themselves

Imagine you are building a small library system for a campus reading room — call it Pustaka. In a purely procedural style, you might store the collection as a handful of parallel arrays: one array of titles, one of authors, one of ISBNs, and one boolean array recording whether each book is currently on loan. Every operation — borrowing a book, returning it, renewing it, printing the catalog — is a free function that receives all four arrays as parameters and an index telling it which book to touch.

This works, at first. The trouble starts as the program grows. Nothing in the language stops any function, anywhere in a thousand-line program, from reaching into the onLoan array and setting an entry to true without ever checking whether that book was actually available, or forgetting to update timesRenewed when a loan is renewed, or updating one array but not its three parallel siblings after a bug is fixed in only one function. There is no single place in the code that guarantees "a book's data can only change in ways that make sense for a real book" — that responsibility is smeared across every function that happens to touch those arrays, and every new function added to the program is one more place that guarantee can quietly break.

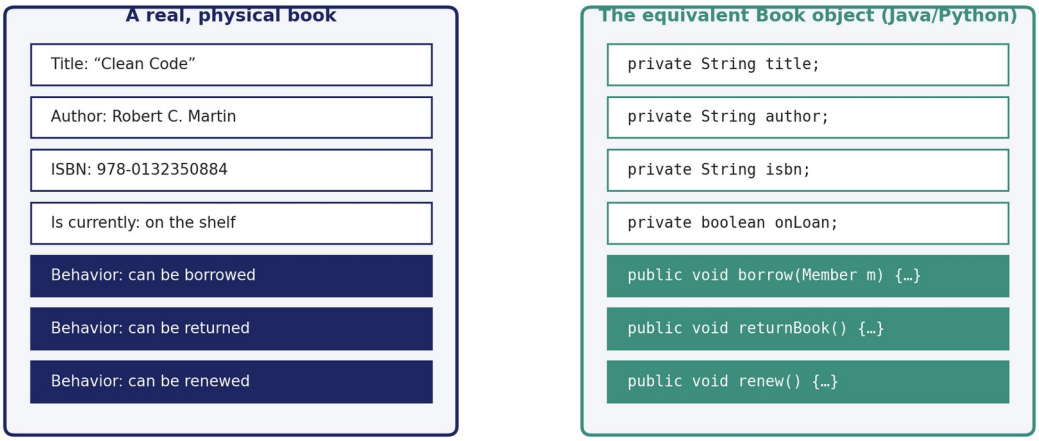
> **"Just be careful" does not scale**
>
> A common first reaction is: "the programmer just needs to be careful not to misuse the arrays." This works for a 200-line student assignment written by one person in one sitting. It reliably fails for any program maintained by more than one person, touched more than once, or larger than a few hundred lines — which describes essentially all real, employed software. OOP does not make careless mistakes impossible, but it gives the language itself tools to make many classes of mistakes impossible to compile, rather than merely asking programmers to remember not to make them.

1.3 Objects: Bundling Data and Behavior Together

Object-oriented programming's core move is to stop treating a book's data (title, author, ISBN, loan status) and a book's behavior (borrow, return, renew) as separate things scattered across the program. Instead, a class defines a blueprint — Book — that bundles the data fields and the behavior methods into one unit, and an object is one concrete instance of that blueprint, exactly the way one physical book on the shelf is one concrete instance of the general idea "a book."

Anatomy of an Object: A Real Book and a Book Object, Side by Side

An object is just a bundle of data (fields) and behavior (methods) that belong together — exactly like a real-world thing.



Fields hold the book's DATA. Methods are the book's BEHAVIOR. Bundling both together, and hiding the fields behind methods, is called encapsulation — the very first OOP idea this chapter introduces.

Figure 1.2 — A real book and a Book object hold the same information; the object additionally exposes only the operations (borrow, returnBook, renew) that make sense to perform on it.

Encapsulation, previewed

Marking a field private (as every field in Figure 1.2's Book object is) means no code outside the class can read or write that field directly — the only way to affect onLoan is to call borrow(), returnBook(), or renew(), and those methods can refuse a request that does not make sense (for example, borrowing a book that is already on loan). This bundling-plus-hiding is called encapsulation, and it is the first of the four pillars of OOP this course covers: encapsulation (Chapter 2), inheritance (Chapter 9), polymorphism (also Chapter 9), and abstraction via interfaces and abstract classes (Chapter 7).

1.4 Procedural vs. Object-Oriented: The Same Problem, Two Mental Models

Figure 1.3 places the two styles side by side for the exact same Pustaka library problem. The procedural version on the left keeps four parallel arrays and a set of free functions that must all be given the right array and the right index every time. The object-oriented version on the right defines one Book class; each Book object privately owns its own title and onLoan flag, and the only way to change onLoan is to call a method that Book itself defines and can refuse.

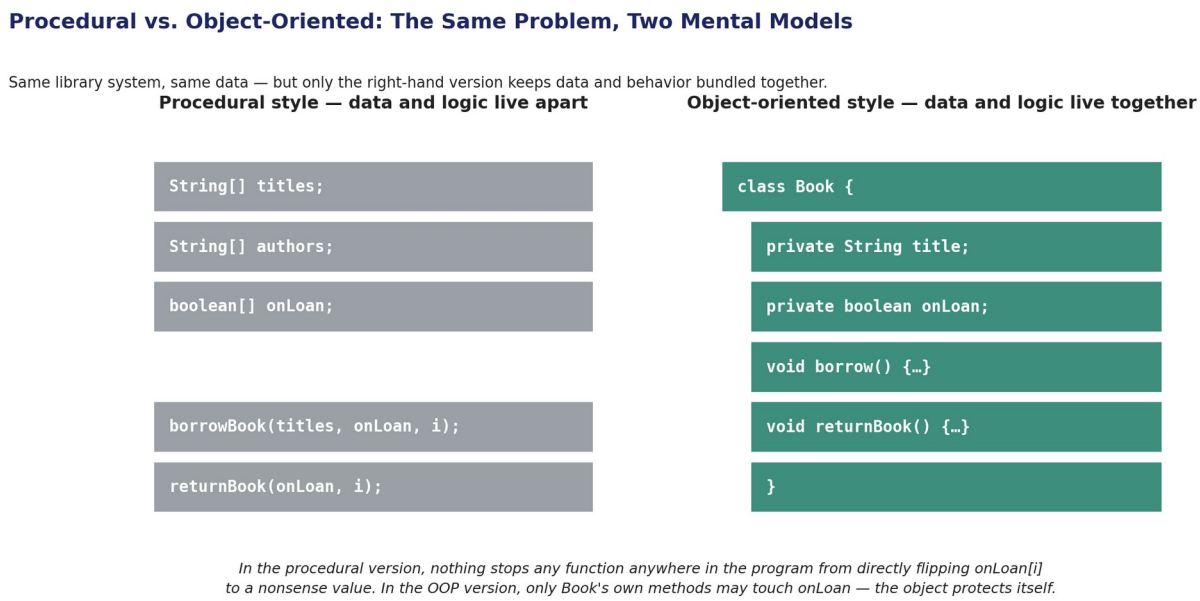


Figure 1.3 — The procedural version can be corrupted from anywhere in the program. The object-oriented version protects its own data by construction, not by convention.

Neither style can do anything the other one fundamentally cannot — OOP is not a new kind of computation, it is a different way of organizing the same computations for clarity, safety, and change-tolerance. As programs grow past a few hundred lines and past a single author, that organizational difference becomes the difference between a codebase you can safely extend for years and one where every change risks breaking something far away that you never noticed depended on the old, unprotected data.

1.5 A First Look at the Four Pillars

This course is organized around four ideas that recur constantly in professional Java and Python code. This section previews each one in one sentence; each gets a full chapter later.

- Encapsulation (Chapter 2) — bundling data with the methods that operate on it, and hiding the data behind those methods so it can only change in valid ways.
- Abstraction via interfaces & abstract classes (Chapter 7) — describing what an object can do ("this can be borrowed") without committing to exactly how, so different kinds of objects can be used interchangeably wherever that behavior is expected.
- Inheritance (Chapter 9) — letting one class reuse and specialize the fields and methods of another, so a ReferenceBook and a Textbook can share almost everything a general Book already defines.
- Polymorphism (Chapter 9) — calling the same method name on objects of different concrete types and getting behavior appropriate to each type, without the caller needing to know which type it is talking to.

Why this order?

Every professional Java or Python codebase you will ever read leans on all four pillars simultaneously, so the order in which a course introduces them is a teaching choice, not a fact about the language. This course teaches encapsulation first because the other three pillars are, in a real sense, built on top of it: you cannot meaningfully talk about specializing an object's behavior (inheritance/polymorphism) or describing behavior abstractly (interfaces) until you first accept that an object's data should be hidden behind its own methods.

1.6 Setting Up Your Java Development Environment

Every example in this book is written for a current Long-Term-Support (LTS) release of Java. As of August 2026, Oracle's published roadmap lists Java 8, 11, 17, 21, and 25 as LTS releases, with Java 25 (general availability September 2025) the newest — Oracle has stated it intends to ship a new LTS every two years going forward, with Java 29 planned for September 2027. This book's own code was written against, compiled, and run on JDK 21 LTS; everything shown compiles and runs unchanged on JDK 25 as well, since none of this course's material depends on a feature removed or changed between those releases. Do not install JDK 8 to follow this course, even if you find old tutorials that assume it — JDK 8 exited free public updates years ago, and several APIs this book uses (records, pattern matching for switch, sealed classes referenced in later chapters) either do not exist or behave differently on it.

Tool	Recommendation	Why
JDK	Eclipse Temurin (Adoptium) or Oracle JDK, version 21 or 25 LTS	Free, production-grade OpenJDK builds; either LTS line receives updates for years, unlike a short-term release.
IDE — primary	IntelliJ IDEA Community Edition (free)	The most widely used Java IDE in industry today; excellent refactoring tools, built-in Git support, and a gentler project-setup flow than Eclipse for a first course.
IDE — alternative	Eclipse IDE for Java Developers	This course's original materials used Eclipse; it remains a fully capable, free, widely taught choice if your lab or campus already standardizes on it.
Build tool (later chapters)	Maven or Gradle	Introduced when a chapter needs a dependency beyond the standard library (e.g. a JUnit test or a JavaFX GUI in Lecture 11).

You do not have to pick the "one true" IDE

Every code listing in this book is plain .java source compiled with the standard javac compiler — it will compile and run identically whichever IDE (or no IDE at all, from a terminal) you use to edit it. Pick whichever tool your class, employer, or lab requires; nothing in this course depends on IDE-specific project files.

Once your JDK is installed, verify it from a terminal before opening any IDE — this catches a missing or misconfigured installation before it can confuse you inside a more complex tool:

```
$ java -version
openjdk version "21.0.10" 2026-01-20
OpenJDK Runtime Environment ...

$ javac -version
javac 21.0.10
```

1.7 Worked Example: Your First Java Class

This course's worked examples follow a five-phase framework — Understand, Abstract, Design, Analyze, Implement — for every non-trivial problem, starting right now with the smallest possible one: modeling a single book in the Pustaka library.

Phase 1 — Understand

```
Problem: a librarian needs to track, for each book, its title, author,
and ISBN, plus whether it is currently on loan, and support borrowing,
returning, and renewing (up to 2 times per loan) that book.
```

Phase 2 — Abstract

```
Strip away everything specific to "library" and "book" and notice the
general shape: some DATA that must never become internally nonsensical
(e.g. "on loan" but "renewed -3 times"), plus a small set of BEHAVIORS
that are the only sanctioned way to change that data.
```

Phase 3 — Design

```
Fields (private, so only this class's own methods can touch them):
  title, author, isbn : String   (never change after creation)
  onLoan              : boolean (starts false)
  timesRenewed        : int     (starts 0, resets on each new borrow)
Methods (public — the ONLY way outside code may affect this object):
  borrow()      -> boolean   returns false if already on loan
  returnBook()  -> void
  renew()       -> boolean   returns false if not on loan or already
                             renewed twice
  toString()    -> String    human-readable summary, for printing
```

Phase 4 — Analyze

```
Correctness argument: onLoan can only become true inside borrow(), and
only when it was false; it can only become false inside returnBook().
timesRenewed can only increase inside renew(), and only up to 2, and is
reset to 0 every time borrow() succeeds. Because every field is private,
NO code anywhere else in the program can violate these rules even by
accident -- the compiler itself refuses code that tries.
```

Phase 5 — Implement

```
public class Book {
    private final String title;
    private final String author;
    private final String isbn;
    private boolean onLoan;
    private int timesRenewed;

    public Book(String title, String author, String isbn) {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
        this.onLoan = false;
        this.timesRenewed = 0;
    }

    public boolean borrow() {
        if (onLoan) { return false; }
        onLoan = true;
        timesRenewed = 0;
        return true;
    }

    public void returnBook() { onLoan = false; }

    public boolean renew() {
        if (!onLoan || timesRenewed >= 2) { return false; }
        timesRenewed++;
        return true;
    }

    @Override
    public String toString() {
        String status = onLoan
            ? "ON LOAN (renewed " + timesRenewed + "x)"
            : "on the shelf";
        return String.format("\"%s\" by %s [ISBN %s] - %s",
            title, author, isbn, status);
    }
}
```

A class by itself does nothing until something creates an object from it — Java calls this instantiation, using the new keyword. The following driver program (a separate class holding only a main method) creates a Book object and exercises its behavior; it is a shortened excerpt for the page — the complete file, which also creates a second book and demonstrates renew() and returnBook(), ships as Code/Java/Chapter01/LibraryDemo.java alongside this book, and its full output is what Appendix 1.A verifies:

```java
public class LibraryDemo {
    public static void main(String[] args) {
        Book cleanCode = new Book("Clean Code",
            "Robert C. Martin", "978-0132350884");

        System.out.println(cleanCode);
        cleanCode.borrow();
        System.out.println(cleanCode);
        boolean again = cleanCode.borrow();
        System.out.println("Second borrow allowed? " + again);
    }
}
```

Trace it yourself before reading on

What does the program above print on its third line? Walk through it: the first println shows the freshly-constructed book (on the shelf); borrow() then flips onLoan to true; the second println reflects that; and the second call to borrow() — on a book that is already on loan — must return false, because Phase 4's correctness argument guarantees onLoan cannot become true a second time without a returnBook() in between. If your trace disagrees with "Second borrow allowed? false", re-read borrow()'s single if statement.

1.8 Compiling and Running Your Program

Java is a compiled language: the javac compiler translates your .java source files into .class files of JVM bytecode, and the java launcher then runs that bytecode on the Java Virtual Machine. Both files, Book.java and LibraryDemo.java, must be compiled before LibraryDemo can run, because LibraryDemo's source refers to the Book type:

```
$ javac Book.java LibraryDemo.java
$ java LibraryDemo
```

If you are using an IDE such as IntelliJ IDEA or Eclipse, the IDE runs these same two commands for you behind a "Run" button — understanding what actually happens underneath that button will make IDE error messages far less mysterious throughout this course.

1.9 OOP in Practice: Profitina

About this box

Throughout this book, boxes like this one connect course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author. These boxes describe WHERE and WHY a concept is used in a real, running system — never Profitina's proprietary business logic or full source code, which remain confidential. See the companion document "Profitina: A Non-Confidential Technical Overview" for the full picture.

Profitina's own product (app.profitina.com) is written in Python, not Java — but the OOP idea this chapter introduces, encapsulation, applies identically regardless of which language expresses it. Profitina's backend defines classes such as a scan-result object that bundles a business's visibility score together with the methods that are allowed to update it, rather than exposing that score as a bare number any part of the codebase could overwrite. The self-hosted AI models Profitina runs (Qwen2.5

for long-form analysis, Qwen3:1.7b for fast classification, both via Ollama, entirely on Profitina's own server) are themselves wrapped behind Profitina's own classes and methods — the rest of the application never reaches past those objects to poke at model internals directly. This is exactly the Book-class discipline of Section 1.7, scaled up from a teaching example to a live product used by real businesses today.

You will see this same pattern in every later chapter: whatever Java or Python idea a chapter introduces, this course will show you precisely where an equivalent idea shows up inside Profitina's own real, live, production codebase — always at the level of "here is the shape of the idea being used," never at the level of proprietary implementation detail.

1.10 Chapter Summary and Key Takeaways

- Procedural programs that scatter data across parallel arrays have no single place enforcing that the data stays sensible — any function anywhere can corrupt it.
- An object bundles data (fields) with the behavior (methods) that is allowed to change that data; encapsulation means hiding the fields so only the object's own methods can touch them.
- This course's four pillars — encapsulation, abstraction, inheritance, polymorphism — will each get a dedicated chapter, in that order, because each later pillar builds on the discipline the earlier ones establish.
- Java is compiled: javac produces .class bytecode from .java source, and java runs that bytecode on the JVM. Use JDK 21 or 25 (both current LTS releases as of 2026), and IntelliJ IDEA or Eclipse as your IDE.
- The Five-Phase framework (Understand, Abstract, Design, Analyze, Implement) used for Book in this chapter will be used for every non-trivial worked example in this course.

"I made up the term 'object-oriented', and I can tell you I did not have C++ in mind." — the language syntax you will learn in this course is secondary to the mental model this chapter introduces: programs as communicating objects, each responsible for protecting its own data.

1.11 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Lecture 2.

1. In your own words, explain why "the programmer should just be careful" is not considered a satisfactory answer to the data-corruption problem described in Section 1.2.
2. Book's timesRenewed field is private. Write one sentence explaining what would go wrong for Phase 4's correctness argument if it were public instead.
3. Modify (on paper, or actually in your IDE) the Book class to add a method extendDueDate(int days) that is only allowed to succeed if the book is currently on loan. Where in the class must this rule be enforced, and why can it not be enforced anywhere else?
4. Figure 1.3 shows a procedural version with four parallel arrays. Suppose a new field, dueDate, needs to be added. List every place in the procedural version that must change, versus every place in the object-oriented version that must change.

5. Which of the four pillars (encapsulation, abstraction, inheritance, polymorphism) does this chapter's Book example demonstrate? Which three does it deliberately NOT yet demonstrate, and why (hint: re-read Section 1.5)?

Appendix 1.A — Compilation and Execution Verification Log

The complete Book.java and LibraryDemo.java files (Code/Java/Chapter01/, provided alongside this book — a superset of the shortened excerpt in Section 1.7, exercising a second book plus renew() and returnBook() as well) were compiled and executed on OpenJDK 21.0.10 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ javac Book.java LibraryDemo.java
(no errors)

$ java LibraryDemo
--- Pustaka: starting state ---
"Clean Code" by Robert C. Martin [ISBN 978-0132350884] - on the shelf
"The Pragmatic Programmer" by Hunt & Thomas [ISBN 978-0135957059] - on the shelf

--- A student borrows Clean Code ---
Borrow succeeded? true
"Clean Code" by Robert C. Martin [ISBN 978-0132350884] - ON LOAN (renewed 0x)

--- Someone else tries to borrow the SAME copy ---
Borrow succeeded? false (already on loan, so the object refuses)

--- The student renews, then returns it ---
"Clean Code" by Robert C. Martin [ISBN 978-0132350884] - ON LOAN (renewed 1x)
"Clean Code" by Robert C. Martin [ISBN 978-0132350884] - on the shelf
```

References

[1] A. Kay. Email to the Squeak developers' mailing list, 2003 (widely quoted as "I made up the term object-oriented, and I can tell you I did not have C++ in mind").

[2] Oracle Corporation. "Java SE Support Roadmap." https://www.oracle.com/java/technologies/java-se-support-roadmap.html (accessed August 2026) — confirms Java 8, 11, 17, 21, and 25 as LTS releases, Java 25 GA September 2025.

[3] R. C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008 (the source of this chapter's running example title).

[4] JetBrains. "IntelliJ IDEA." https://www.jetbrains.com/idea/ (Community Edition, accessed August 2026).

[5] Eclipse Foundation. "Eclipse IDE for Java Developers." https://www.eclipse.org/downloads/packages/ (accessed August 2026).

CHAPTER 2

Anatomy of a Class: Fields, Constructors & Encapsulation

Chapter 1 told you an object bundles data with behavior. This chapter opens the hood: exactly which parts of a class declare that data, exactly which part builds a new object, and exactly how much of that data the outside world is allowed to see.

Learning Objectives — after this chapter you will be able to:

(1) Identify a class's three kinds of members — fields, constructors, methods — and state what each is for. (2) Write a constructor, and explain what "this" refers to inside one. (3) Overload a constructor and use this(...) to delegate from one constructor to another, avoiding duplicated field-assignment logic. (4) Write getters that expose an object's fields for reading without allowing outside code to corrupt them. (5) Name Java's four access modifiers, state each one's visibility rule, and explain what Python uses instead. (6) Extend the Chapter 1 Book class with a new field, additional constructors, and full getter coverage, then compile, run, and verify it yourself.

2.1 Picking Up Where Chapter 1 Left Off

Chapter 1 introduced the single idea every later chapter builds on: an object bundles data (fields) with the behavior (methods) allowed to change that data, and hides the fields behind that behavior. You saw one class, Book, with three fields and four methods, and you compiled and ran it. This chapter does not introduce a new idea so much as it takes that same Book class apart piece by piece and looks closely at each of its three kinds of members — fields, constructors, and methods — starting with the one member kind Chapter 1 used constantly but never explained: the constructor.

Anatomy of a Class: Fields, Constructors, and Methods

Every Java or Python class you write has (at most) these three kinds of members, in this conventional order.

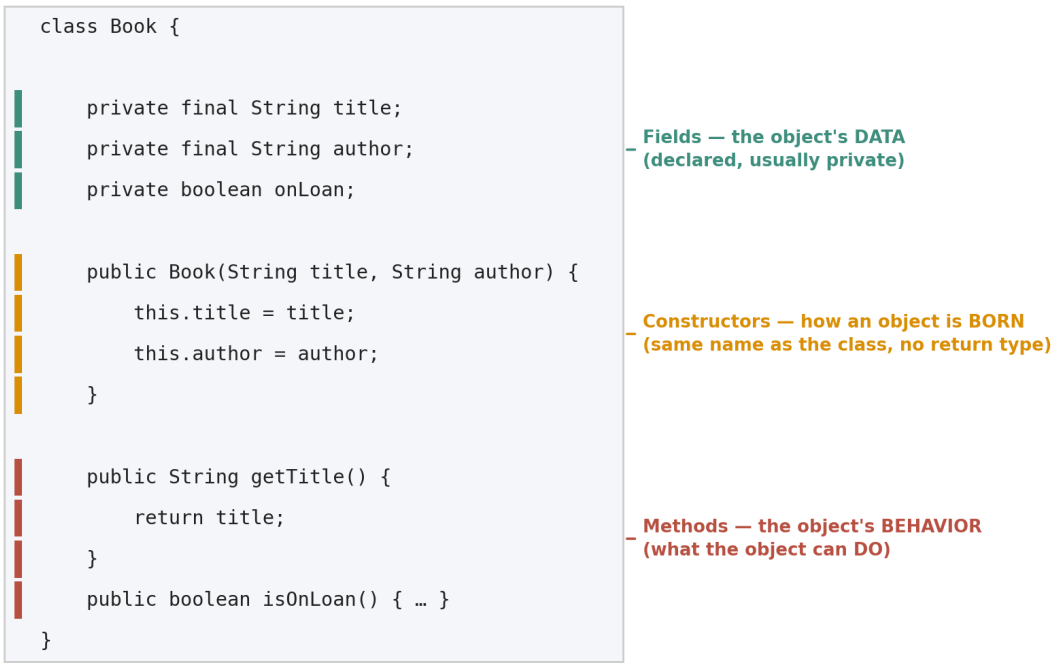


Figure 2.1 — Every class you write has (at most) these three kinds of members, in this conventional order: fields, then constructors, then methods.

2.2 Fields: An Object's Data, Formally

A field is a variable declared directly inside a class body (not inside any method) — it is where an object's data lives. Chapter 1's Book declared three fields: title, author, and isbn, all of type String, plus onLoan (boolean) and timesRenewed (int). Every object built from the Book class gets its own private copy of these five fields; two different Book objects never share the same title variable, even though both are declared by the exact same three-line piece of source code (Figure 2.1).

Notice that title, author, and isbn were declared private final. private restricts who may read or write the field directly (Section 2.6 covers this in full); final adds a second, independent restriction — once a final field is assigned a value inside a constructor, it can never be reassigned for the lifetime of that object. Marking a field final is how Java lets you promise, and have the compiler enforce, "this piece of data cannot change after the object is built." onLoan and timesRenewed are deliberately NOT final, because their whole purpose is to change as the book is borrowed, renewed, and returned.

Python has no field keyword and no final keyword

Python attributes come into existence the moment you assign self._title = title inside __init__ — there is no separate field-declaration syntax to learn, and no compiler-enforced final. If a Python class wants to signal "this attribute should not be reassigned after construction," it relies on the same convention discipline covered in Chapter 1: a leading underscore, a @property with no matching @x.setter, and programmer discipline. Chapter 2's Python code follows this convention throughout.

2.3 Constructors: How Objects Are Born

A class by itself is only a blueprint; nothing exists until code calls new (Java) or calls the class name like a function (Python), and that call always runs a constructor. A constructor's job is singular: take whatever raw values the caller supplies and use them to bring one new object into existence in a valid, fully-initialized state — never a half-built object with some fields set and others still holding garbage values.

In Java, a constructor is a method-shaped block with three unusual properties: it has no return type (not even void), its name is exactly the class's name, and it runs its body exactly once, at the moment new Book(...) executes. Inside a constructor, the keyword this refers to "the object currently being built" — it exists specifically to resolve the common naming collision where a constructor's parameter (title) and the field it is meant to initialize (also called title) share the same name: this.title = title unambiguously means "the field belonging to this object gets the value of the parameter."

Forgetting this. is a classic first bug

If a constructor parameter and a field share a name and you write title = title; instead of this.title = title;, Java resolves the bare name title to the closest-in-scope declaration — the parameter — so the statement assigns the parameter to itself and the field is left completely unset (null, for a String field). This compiles without error and fails silently, which is exactly why it is worth tracing through carefully the first time you see it. Python sidesteps this specific trap because self.title and title are never ambiguous — self. is required to reach the attribute, so there is no bare name to accidentally shadow.

2.4 Constructor Overloading and this(...) Chaining

Real callers do not always have the same information available. Sometimes you know a book's publication year; sometimes the year is simply not recorded anywhere you have access to. Java lets one class define multiple constructors as long as their parameter lists are distinguishable (different number or different types of parameters) — this is called constructor overloading, and the compiler picks the right one to run based on how many arguments, and what type, you pass to new Book(...) (Figure 2.2).

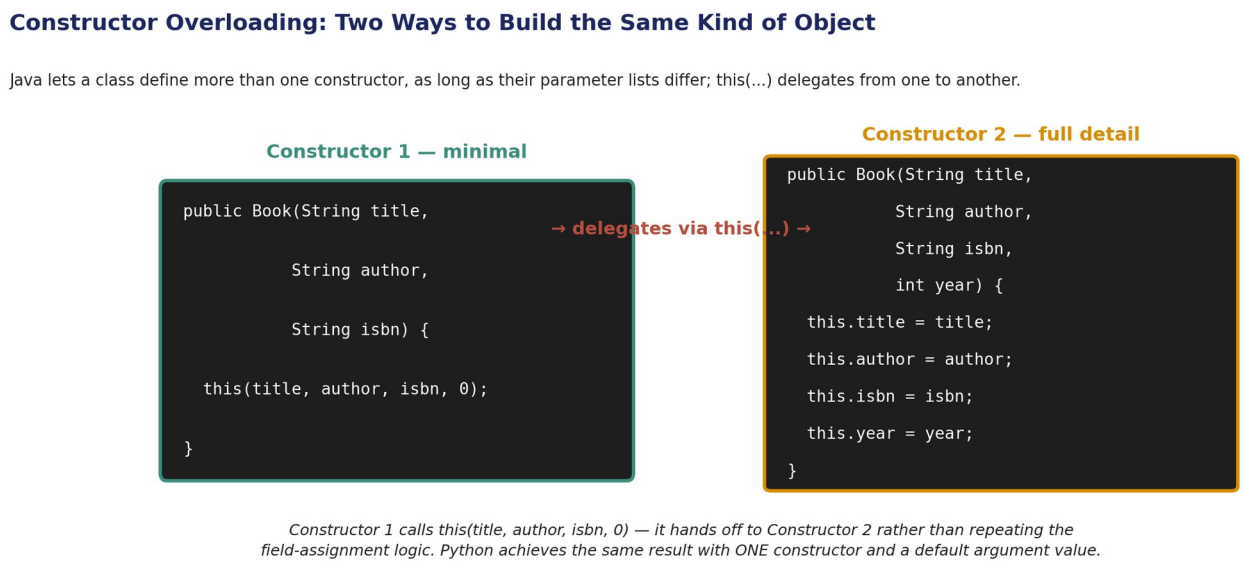


Figure 2.2 — Constructor 1 (three arguments) delegates to Constructor 2 (four arguments) via this(title, author, isbn, 0), so the field-assignment logic is written exactly once.

The one-line body this(title, author, isbn, 0); inside the three-argument constructor is a constructor call, not a field assignment — it must be the very first statement in the constructor body, and it hands the work off to the four-argument constructor, supplying 0 as a stand-in value meaning "publication year not recorded." Without this(...) chaining, both constructors would need to repeat all five field-assignment lines, and a future bug fix to one copy could easily be forgotten in the other.

> **Constructor overloading, Python style: one constructor, default arguments**
>
> Python method signatures do not support true overloading the way Java's do — there is exactly one __init__ per class. Python instead achieves the exact same outcome with a single constructor and a default parameter value: def __init__(self, title, author, isbn, publication_year=0). Calling Book(t, a, i) and Book(t, a, i, 2008) both work, and both run the same __init__ body — Python has no separate 'minimal' constructor to keep in sync with a 'full' one, so there is no this(...)-style chaining to learn in the first place.

2.5 Getters: Controlled Read Access

Marking a field private (Java) or leading-underscore (Python) solves the write-side of encapsulation — nothing outside the class can corrupt the field by assigning it a nonsensical value. But it also, as an unavoidable side effect, blocks the outside world from simply reading the field's current value, which is very often something outside code legitimately needs (a catalog page needs to display a book's title; a search feature needs to check its author). A getter is a small public method whose entire job is to return one field's current value, restoring read access without reopening write access.

A getter is deliberately the simplest possible kind of method: it takes no parameters, and its body is a single return statement. Java's naming convention prefixes get to the field name (getTitle(), getAuthor()), except for boolean fields, which conventionally use is instead of get (isOnLoan(), not getOnLoan()) — this is a community convention, not a language rule, but nearly every Java codebase you will ever read follows it, and breaking it needlessly confuses every other Java programmer who reads your code.

Java getter	Python @property	Returns
getTitle()	title	the book's title (String / str)
getAuthor()	author	the book's author (String / str)
getIsbn()	isbn	the book's ISBN (String / str)
getPublicationYear()	publication_year	the year, or 0 if unrecorded (int)
isOnLoan()	is_on_loan	whether the book is currently borrowed (boolean / bool)
getTimesRenewed()	times_renewed	how many times the current loan was renewed (int)

A getter returning a String or int is still safe, even without "copying" it

A natural worry: if getTitle() just returns the field directly, can't the caller who receives it turn around and corrupt Book's internal state through that returned reference? For String, int, and boolean specifically, no — these types are immutable in both Java and Python, meaning there is no method on a String or an int that could change the characters or digits it already holds. The caller receives a value they can read, copy, or discard, but never a handle that lets them reach back inside Book and mutate its private field. (This question becomes far more interesting once a getter returns a mutable type, such as a list of loans — a problem this course revisits directly in Chapter 10, Generics & Collections.)

2.6 Access Modifiers: Java's Four Levels vs. Python's Convention

Section 2.2 and Chapter 1 have both used the word private repeatedly without fully cataloging its alternatives. Java defines four levels of visibility a field, constructor, or method can be given, and Section 2.5's getters exist specifically to open a narrow, controlled hole through the strictest of the four (Figure 2.3).

Access Modifiers at a Glance: Java's Four Levels vs. Python's Convention

Java's compiler enforces visibility; Python signals the same intent through naming convention and trusts the programmer.

Java modifier	Visible from	Python equivalent
`private`	Class itself only	`_single_underscore` (convention: internal)
`(package-private)`	Same package only	— (Python has no package-level access control)
`protected`	Package + subclasses	— (rarely needed until Chapter 9, inheritance)
`public`	Anywhere	no underscore (the default)

Java's private is enforced by the compiler — code outside the class cannot even attempt to compile a violation. Python's _leading_underscore is a promise between programmers, not a wall the interpreter builds for you.

Figure 2.3 — Java's four access levels, from narrowest (private) to widest (public), alongside Python's naming-convention equivalent for each.

private is the level this course uses for every field so far, and it is the strictest: visible only inside the class's own body. package-private (Java's default, when no keyword at all is written) opens visibility to every other class in the same package — a middle ground this course does not lean on until later chapters group related classes together. protected extends package-private visibility to subclasses as well, wherever they live — a level whose full usefulness only becomes visible once Chapter 9 introduces inheritance. public, the widest level, is what every getter and every behavior method (borrow(), returnBook(), renew()) in this course uses, since those methods are precisely the controlled doorway the rest of the program is meant to use.

> **public everything defeats the entire point of Chapter 1**
>
> It is technically legal to mark every field public and delete every getter and every access-restricting method — Java will compile this without complaint. Doing so throws away everything Chapter 1 argued for: with every field public, any code anywhere in the program can set onLoan = true directly, completely bypassing borrow()'s check for whether the book is already on loan, reintroducing the exact data-corruption risk Chapter 1's procedural-vs-OOP comparison (Figure 1.3) was built to demonstrate. Encapsulation is not the private keyword by itself; it is private fields deliberately paired with a small, curated set of public methods.

2.7 Worked Example: Extending Book with a Publication Year

This chapter's worked example takes Chapter 1's Book class and extends it: a new publicationYear field, two constructors (demonstrating overloading and this(...) chaining), and a getter for every field.

Phase 1 — Understand

```
New requirement: some part of the program (a future catalog search
feature) needs read access to a book's title, author, isbn, loan
status, and renewal count -- and needs to record a book's
publication year when it is known, without requiring it when it isn't.
```

Phase 2 — Abstract

```
General shape: one new piece of DATA (publicationYear), plus a
controlled READ channel (getters) for every existing piece of data,
plus a way to construct the object whether or not the new data
point is available at construction time.
```

Phase 3 — Design

```
New field:
  publicationYear : int (final; 0 means "not recorded")
Constructors (overloaded, the 3-arg one delegates via this(...)):
  Book(title, author, isbn)              -- year defaults to 0
  Book(title, author, isbn, publicationYear)
Getters (public; the ONLY read channel for private fields):
  getTitle(), getAuthor(), getIsbn(), getPublicationYear(),
  isOnLoan(), getTimesRenewed()   -- all no-argument, single-return
```

Phase 4 — Analyze

Correctness argument: publicationYear is final, so once a Book is constructed it cannot silently change, exactly like title, author, and isbn. Every getter returns an immutable type (String, int, or boolean), so no caller who receives a getter's return value can reach back in and mutate Book's private state through it. The two constructors cannot drift out of sync, because the 3-arg one delegates its entire body to the 4-arg one via this(...) -- there is exactly one place the five fields are actually assigned.

Phase 5 — Implement

```
public class Book {
    private final String title;
    private final String author;
    private final String isbn;
    private final int publicationYear;
    private boolean onLoan;
    private int timesRenewed;

    public Book(String title, String author, String isbn) {
        this(title, author, isbn, 0);
    }

    public Book(String title, String author, String isbn,
                int publicationYear) {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
        this.publicationYear = publicationYear;
        this.onLoan = false;
        this.timesRenewed = 0;
    }

    // borrow(), returnBook(), renew() -- unchanged from Chapter 1

    public String getTitle() { return title; }
    public String getAuthor() { return author; }
    public String getIsbn() { return isbn; }
    public int getPublicationYear() { return publicationYear; }
    public boolean isOnLoan() { return onLoan; }
    public int getTimesRenewed() { return timesRenewed; }
}
```

The driver program below creates two Book objects, one using the four-argument constructor (a known publication year) and one using the three-argument constructor (an unrecorded year, defaulted to 0 via this(...) chaining), then exercises several getters. It is shown here in full, exactly as it appears in Code/Java/Chapter02/LibraryDemo.java, and its complete output is exactly what Appendix 2.A verifies:

```
public class LibraryDemo {
    public static void main(String[] args) {
        Book cleanCode = new Book("Clean Code", "Robert C. Martin",
            "978-0132350884", 2008);
        Book pragmatic = new Book("The Pragmatic Programmer",
            "Hunt & Thomas", "978-0135957059");

        System.out.println(cleanCode);
        System.out.println(pragmatic);

        System.out.println("Clean Code author: "
            + cleanCode.getAuthor());
        System.out.println("Pragmatic Programmer year: "
            + pragmatic.getPublicationYear() + " (0 = unrecorded)");

        cleanCode.borrow();
        System.out.println(cleanCode);
    }
}
```

Trace it yourself before reading on

pragmatic is built with the three-argument constructor, which never mentions a publication year at all. What does pragmatic.getPublicationYear() return, and what does the toString() line print for pragmatic's year, given that? Trace the delegation: the 3-arg constructor calls this(title, author, isbn, 0) — so 0 is exactly the value stored in publicationYear, and toString() (Section 2.7's Phase 5, extended further in the full source file) is written to omit the year bracket entirely whenever publicationYear is 0, rather than printing the misleading number 0 itself.

2.8 Compiling and Running Your Program

Nothing changes about the compile-and-run process Chapter 1, Section 1.8 introduced: javac Book.java LibraryDemo.java compiles both files (still required together, since LibraryDemo's source refers to the Book type), and java LibraryDemo runs the compiled bytecode.

```
$ javac Book.java LibraryDemo.java
$ java LibraryDemo
```

If you extend Book.java yourself while working through this chapter's Review Questions, get in the habit of recompiling BOTH files together with a single javac call, exactly as shown above, rather than compiling only the file you edited — Java will refuse to run LibraryDemo against a stale, outdated Book.class left over from before your edit, and recompiling both together every time removes any chance of that confusion.

2.9 OOP in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential. See the companion document "Profitina: A Non-Confidential Technical Overview" for the full picture.

Profitina's backend defines classes whose constructors mirror exactly the pattern this chapter teaches: a scan-result class, for instance, is built once per completed AI-visibility scan, with every field it needs (the business's score, the scan timestamp, which AI models were consulted) supplied at construction time and then treated as effectively fixed for that object's lifetime — the same discipline this chapter's final publicationYear field demonstrates in miniature. Where later code needs to read a scan result's score to display it on a dashboard, it does so through a getter-equivalent accessor, never by reaching directly into the object's internal fields, for exactly the reason Section 2.5 explains: a controlled read channel that can never become an uncontrolled write channel.

This is also where Profitina's own code relies on Java-equivalent access-modifier discipline even though its backend is written in Python: fields meant to stay internal use Python's leading-underscore convention throughout (Section 2.2's callout), and every one of Profitina's own classes exposes its data exclusively through @property getters, not bare attribute access — the same convention this chapter's Python code follows for Book.

2.10 Chapter Summary and Key Takeaways

- A class has (at most) three kinds of members: fields (the object's data), constructors (how the object is built), and methods (the object's behavior).
- A constructor's job is to bring one object into existence in a valid state; this. (Java) resolves the common naming collision between a constructor parameter and the field it initializes.
- Constructor overloading (Java) lets a class offer multiple ways to build an object; this(...) chaining lets one constructor delegate to another so field-assignment logic is written exactly once. Python achieves the same outcome with one constructor and a default argument value.
- A getter is a small public method that returns one field's value, restoring read access without reopening write access — the field stays private; only a narrow, curated doorway is opened.
- Java has four access levels — private, package-private, protected, public — of increasing visibility; Python signals the same intent through naming convention (leading underscore) rather than compiler enforcement.
- Marking every field public defeats the purpose of encapsulation established in Chapter 1 — private fields deliberately paired with a small set of public methods is what makes an object's correctness arguable at all.

A class is a promise about what can happen to its data, and a constructor is where that promise starts being kept. Everything this chapter added to Book — a new field, two constructors, six getters — exists to keep exactly the same promise Chapter 1 made, just for a slightly richer object.

2.11 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Lecture 3.

1. Book's two constructors both ultimately assign all five fields, but only one of them contains the actual assignment statements. Explain, in your own words, why that is a deliberate design choice rather than an oversight.
2. What would happen — be specific about which line fails and why — if you tried to write publicationYear = publicationYear; after the this(...) call inside Book's three-argument constructor, given that publicationYear is declared final?
3. isOnLoan() is named with an is- prefix rather than get-. Write one sentence explaining the convention this signals, and name one other boolean-returning method in this course's Book class that should follow the same convention (hint: re-check Chapter 1).
4. Suppose a getPublisher() method were added that returns a mutable java.util.List<String> of past publishers rather than a single String. Explain why Section 2.5's INSIGHT box's safety argument would no longer fully apply, and what could go wrong.
5. Write a getRemainingRenewals() method for Book that returns how many renewals are still available (the maximum of 2, minus timesRenewed). Should this new method be public or private, and does adding it require touching any existing field's access modifier? Justify your answer.

Appendix 2.A — Execution Verification Log

The complete Book.java and LibraryDemo.java files (Code/Java/Chapter02/, provided alongside this book) were compiled and executed on OpenJDK 21.0.10 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ javac Book.java LibraryDemo.java
(no error)

$ java LibraryDemo
"Clean Code" by Robert C. Martin [ISBN 978-0132350884, 2008] - on the shelf
"The Pragmatic Programmer" by Hunt & Thomas [ISBN 978-0135957059] - on the shelf
Clean Code author: Robert C. Martin
Pragmatic Programmer year: 0 (0 = unrecorded)
"Clean Code" by Robert C. Martin [ISBN 978-0132350884, 2008] - ON LOAN (renewed
0x)
```

References

[1] Oracle Corporation. "The Java Tutorials — Classes and Objects." https://docs.oracle.com/javase/tutorial/java/javaOO/ (accessed August 2026).

[2] Oracle Corporation. "Java SE Support Roadmap." https://www.oracle.com/java/technologies/java-se-support-roadmap.html (accessed August 2026) — confirms Java 8, 11, 17, 21, and 25 as LTS releases, Java 25 GA September 2025.

[3] R. C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008 (the source of this chapter's running example title).

[4] JetBrains. "IntelliJ IDEA." https://www.jetbrains.com/idea/ (Community Edition, accessed August 2026).

[5] Eclipse Foundation. "Eclipse IDE for Java Developers." https://www.eclipse.org/downloads/packages/ (accessed August 2026).

CHAPTER 3

Primitive Types, Control Flow, Strings & Debugging Tools

Chapters 1 and 2 built and extended one class. This chapter steps back to the language fundamentals every method body actually runs on: Java's eight primitive types, the two ways to branch, three ways to loop, why Strings behave the way they do, and how to find out what your program is actually doing when it isn't doing what you expected (Figure 3.1).

Learning Objectives — after this chapter you will be able to:

(1) Name Java's eight primitive types and state each one's size and default value. (2) Write if/else if/else chains and both forms of switch (classic colon-form and the modern Java 14+ expression form) to encode branching logic. (3) Choose correctly among for, while, and do-while loops, and explain the enhanced for-loop's relationship to the classic for loop. (4) Explain String immutability and the string pool, and correctly choose between == and .equals() when comparing Strings. (5) Use print debugging, an IDE's breakpoints, and assert statements to diagnose an incorrect program. (6) Extend Pustaka's Book class with a fine-calculation method and a renewal-status method, then compile, run, and verify it yourself.

3.1 Picking Up Where Lecture 2 Left Off

Chapter 2 opened up the Book class and looked at its three kinds of members: fields, constructors, and methods. Every method you wrote in Chapters 1 and 2 -- borrow(), renew(), the getters -- has a body, and every one of those bodies is built out of exactly the same small set of ingredients: values of a handful of primitive types, decisions (if this, then that), repetition (do this several times), and text (String). This chapter looks directly at those ingredients, using Book itself as the running example wherever it helps, and closes with the practical skill every one of the last two chapters has assumed you already had: how to find out why a program is doing something other than what you expected.

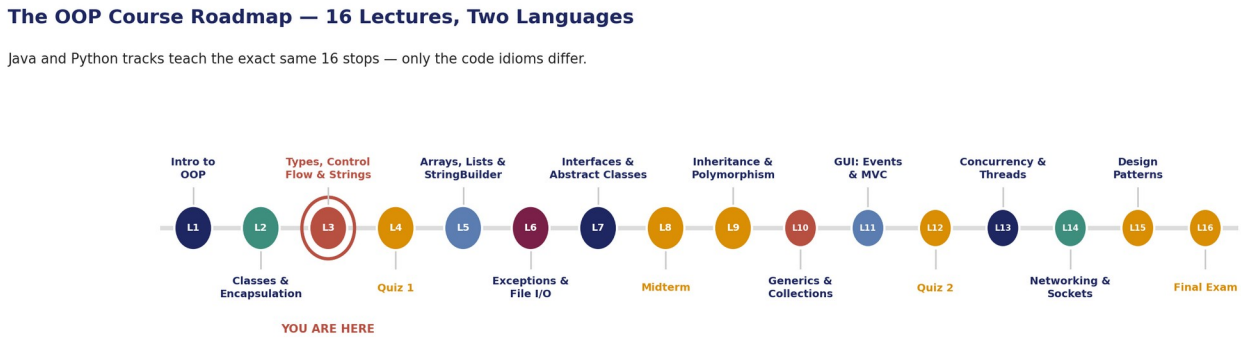


Figure 3.1 — The 16-lecture OOP roadmap. This chapter is Lecture 3, the last stop before Quiz 1 (Lecture 4) checks Lectures 1 through 3 together.

3.2 Java's Eight Primitive Types

Every value in Java is either an object reference (Chapters 1-2's entire subject so far) or one of exactly eight primitive types. A primitive is not an object: it has no methods, no fields of its own, and a variable

of primitive type holds the actual value directly rather than a reference to an object elsewhere in memory. Book already uses three of the eight: int (publicationYear, timesRenewed) and boolean (onLoan) (Figure 3.2).

Java's Eight Primitive Types, at a Glance

Every value that is not an object reference is one of exactly these eight types.

Type	Size	Range / Values	Default
byte	1 byte	-128 to 127	0
short	2 bytes	-32,768 to 32,767	0
int	4 bytes	~ -2.1 billion to 2.1 billion	0
long	8 bytes	~ ±9.2 × 10^18	0L
float	4 bytes	~7 significant decimal digits	0.0f
double	8 bytes	~15 significant decimal digits	0.0
char	2 bytes	single UTF-16 code unit	'\u0000'
boolean	1 bit (JVM-dependent)	true or false	false

int and double are the two workhorses -- reach for byte/short/float only when a specific size or memory constraint calls for them, and char only for a single character, never for general text (use String for that).

Figure 3.2 — Java's eight primitive types, with each one's size, range, and default value. int and double, highlighted, are the two you will reach for by far the most often.

Two practical rules cover the overwhelming majority of real code. First, default to int for whole numbers and double for numbers with a fractional part; reach for byte, short, or float only when a specific memory or precision constraint calls for it, which in an introductory course is rare. Second, char holds exactly one UTF-16 code unit and is not a substitute for text in general -- a book's title is a String (Chapter 1), never a char.

Comparing float or double with == is a classic silent bug

0.1 + 0.2 == 0.3 evaluates to false in Java (and in nearly every other language using the same IEEE 754 floating-point representation), because 0.1 and 0.2 cannot be represented exactly in binary. Never compare float or double values for exact equality; compare Math.abs(a - b) against a small tolerance instead. This chapter's calculateFine method deliberately returns a double for a monetary amount specifically to demonstrate the pattern responsibly -- Section 3.7 shows fine amounts being printed and compared by inspection, never checked with ==.

3.3 Control Flow: if/else if/else and switch

An if statement's condition must be a boolean expression -- unlike some languages, Java never silently treats an int or a String as true or false. Chaining else if lets a decision cascade through several mutually exclusive possibilities, ending in an optional else that catches everything the earlier branches did not.

switch offers a second way to make the same kind of decision when every branch compares one variable against a small set of constant values. Java's original switch (colon-form, case X: ... break;) is notorious for one specific bug: forgetting break lets execution silently 'fall through' into the next case.

Java 14 introduced an expression form of switch (case X -> value;) that has no break to forget and cannot fall through -- each arm is a single expression, and switch itself becomes a value you can return or assign directly (Figure 3.3).

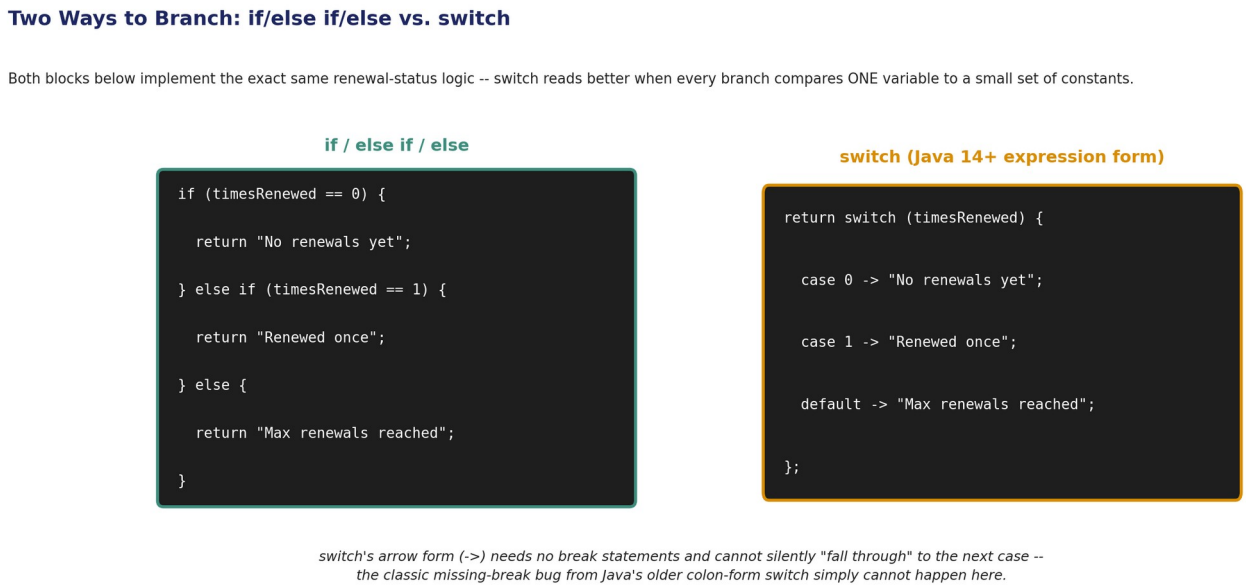


Figure 3.3 — The same renewal-status decision, written as if/else if/else and as a Java 14+ switch expression. Both are correct; switch communicates "one variable, several constant cases" more directly.

> **When to prefer switch over if/else if/else**
>
> Reach for switch when every branch tests the same single variable against specific constant values (an int, a String, or an enum) -- exactly renewalStatusLabel()'s situation in Section 3.7. Prefer if/else if/else when branches test different variables, or test ranges/inequalities rather than exact-match constants -- exactly calculateFine()'s situation, which compares daysOverdue against thresholds (<= 7, <= 30), not exact values switch cannot express directly.

3.4 Loops: for, while, and do-while

Java offers three loop forms, and Chapter 1 already used a fourth (the enhanced for-loop, for (Type x : collection)) without naming it. The classic for loop -- for (initialization; condition; update) -- is the right choice whenever you know in advance how many times you intend to iterate, or need an index: for (int i = 0; i < 5; i++) runs its body exactly five times, with i taking the values 0 through 4.

while (condition) { ... } checks its condition before every iteration, including the first -- it may run zero times if the condition starts out false. do { ... } while (condition); checks its condition after every iteration, so its body always runs at least once, useful for prompt-and-validate patterns ("ask the user for input, and keep asking until it's valid") where you need to ask before you have anything to check.

> **Off-by-one errors are the single most common loop bug**
>
> for (int i = 0; i <= 5; i++) runs SIX times (0 through 5 inclusive), not five -- a one-character difference (<= instead of <) that silently produces one extra iteration. When a loop's iteration count looks wrong, checking the exact boundary condition (< vs <=, and whether the starting index is 0 or 1) is the very first thing to check, before suspecting anything more exotic.

3.5 Strings: Immutability and Comparison

A Java String, once created, can never be changed -- every String method that appears to modify a String (toUpperCase(), trim(), replace(...)) actually returns a brand-new String, leaving the original untouched. This is String immutability, and it has a direct, visible consequence: to conserve memory, Java maintains a string pool of literal Strings, and two String literals with identical text refer to the exact same pooled object rather than two separate ones.

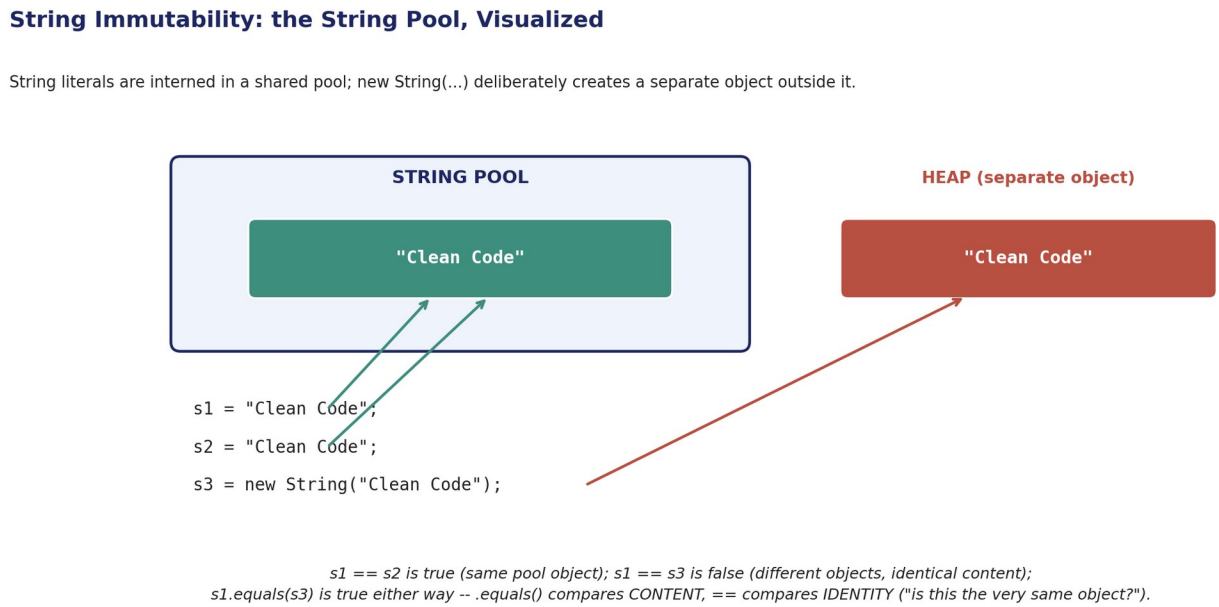


Figure 3.4 — s1 and s2, both String literals with the same text, refer to the SAME pooled object. s3, built with new String(...), deliberately refers to a separate object with identical content.

This is exactly why Java has two different comparison operations that can both look correct at first glance. == compares references -- "are these two variables pointing at the very same object?" -- while .equals() compares content -- "do these two objects hold the same value?" For Strings specifically, always use .equals() (or Objects.equals(a, b) if either might be null) to compare content; == on Strings compiles and runs without error but silently compares identity, which happens to match for pooled literals and just as often does not match for Strings built any other way (from user input, from file contents, from new String(...)).

> **The Scanner input bug: if (userInput == "exit") almost never works**
>
> Text read via Scanner.nextLine() (or any other real input source) is never pulled from the string pool -- it is always a freshly built String object, even when its content exactly matches a literal elsewhere in your code. if (userInput == "exit") therefore compares two different objects and is false even when the user typed exactly "exit" -- one of the most common early Java bugs, and one this section's distinction fully explains rather than leaving as a rule to memorize.

3.6 Debugging Tools: Finding Out What Your Program Is Actually Doing

Every program you have written so far has, at some point, done something other than what you expected. Three tools cover nearly every situation an introductory course encounters, and knowing which one fits a given moment is itself a skill worth building deliberately.

Tool	Best for	Caveat
System.out.println debugging	A quick, immediate check of one or two values, anywhere in the code	Must be removed (or converted to a real logging call) before the code is considered finished
IDE breakpoints (step-over, step-into, watch)	Following a program's exact execution path and inspecting many variables at once, without editing the source	Requires learning the specific IDE's debugger UI; overkill for a one-value check
assert statements	Documenting and checking an internal invariant that should NEVER be false if the code is correct	Disabled by default at runtime -- must run with the -ea JVM flag, or the assertion is silently skipped

Bisection: the fastest way to localize an unknown bug

When a program's output is wrong somewhere in a long sequence of steps but you don't yet know where, print (or set a breakpoint on) the value roughly halfway through the sequence rather than at the very start. If it's already wrong there, the bug is in the first half; if it's still right, the bug is in the second half. Repeating this bisection narrows a hundred-line search down to a handful of lines in about seven checks, the same halving idea Data Structures (a later course) formalizes as binary search.

assert is for YOUR bugs, never for validating user input

assert firstName != null; is appropriate to document "this can never be null if my own code is correct" -- and because assertions are disabled by default in production, code that depends on an assert actually running to reject bad DATA (as opposed to catching a bug in your OWN logic) will silently let that bad data straight through in a real deployment. User input and other untrusted data belong behind an explicit if check (or, from Chapter 6 onward, a thrown exception), never behind assert alone.

3.7 Worked Example: Fine Calculation and Renewal Status for Book

This chapter's worked example adds two small methods to Book: calculateFine(int daysOverdue), demonstrating primitive int/double arithmetic and an if/else if/else tier structure, and renewalStatusLabel(), rewriting the exact same three-way decision already familiar from timesRenewed as a switch expression.

Phase 1 — Understand

```
New requirement: given how many days a book is overdue, compute a fine
amount that increases the longer the book stays out, then caps at a
maximum -- plus a human-readable label for how many times it has been
renewed, rewritten using this chapter's new switch-expression syntax.
```

Phase 2 — Abstract

```
One new BEHAVIOR taking a primitive int and returning a primitive
double (a monotonically increasing, capped fine schedule), plus a
second BEHAVIOR re-expressing existing state (timesRenewed) through
a different but equivalent control-flow construct.
```

Phase 3 — Design

```
PHASE 3 -- DESIGN

New methods (no new fields -- both read existing state or a
parameter):
  calculateFine(int daysOverdue) -> double
    daysOverdue <= 0            -> 0.00
    daysOverdue in 1..7         -> daysOverdue * 0.50
    daysOverdue in 8..30        -> 3.50 + (daysOverdue - 7) * 1.00
    daysOverdue > 30            -> 50.00 (capped)
  renewalStatusLabel() -> String
    switch (timesRenewed) { 0 -> ...; 1 -> ...; default -> ...; }
```

Phase 4 — Analyze

```
Correctness argument: calculateFine's three comparisons (<= 0, <= 7,
<= 30) are mutually exclusive and collectively exhaustive -- every
int falls into exactly one branch, and the fine amount is
non-decreasing as daysOverdue grows, matching the requirement.
renewalStatusLabel's switch expression is exhaustive by construction
(default covers every int not explicitly listed), so it can never
fail to return a value the way an incomplete if-chain silently could.
```

Phase 5 — Implement

```
// fields, constructors, and getters: unchanged from Chapter 2

public double calculateFine(int daysOverdue) {
    if (daysOverdue <= 0) {
        return 0.0;
    } else if (daysOverdue <= 7) {
        return daysOverdue * 0.50;
    } else if (daysOverdue <= 30) {
        return 3.50 + (daysOverdue - 7) * 1.00;
    } else {
        return 50.00;
    }
}

public String renewalStatusLabel() {
    return switch (timesRenewed) {
        case 0 -> "No renewals yet";
        case 1 -> "Renewed once";
        default -> "Max renewals reached";
    };
}
```

The driver below demonstrates both new methods, loops over an array of sample overdue-day counts with an enhanced for-loop (Section 3.4), and closes with the == vs .equals() comparison Section 3.5 warned about, deliberately constructing a non-pooled String with new String(…) to make the difference visible. It is shown here in full, exactly as it appears in Code/Java/Chapter03/LibraryDemo.java, and its complete output is exactly what Appendix 3.A verifies:

```
Book cleanCode = new Book("Clean Code", "Robert C. Martin",
    "978-0132350884", 2008);
System.out.println(cleanCode);

cleanCode.borrow();
cleanCode.renew();
System.out.println("Renewal status: " + cleanCode.renewalStatusLabel());

int[] overdueDaysSamples = {0, 3, 15, 45};
for (int days : overdueDaysSamples) {
    double fine = cleanCode.calculateFine(days);
    System.out.printf("Overdue %2d day(s) -> fine: %.2f%n", days, fine);
}

String isbnFromCatalog = "978-0132350884";
String isbnFromScanner = new String("978-0132350884");
System.out.println("== comparison:        "
    + (isbnFromCatalog == isbnFromScanner));
System.out.println(".equals() comparison: "
    + isbnFromCatalog.equals(isbnFromScanner));
```

Trace it yourself before reading on

After borrow() and one renew() call, timesRenewed is 1 -- what does renewalStatusLabel() return? Trace the fine tiers: 0 days is exactly 0.00 (the boundary belongs to the first branch, daysOverdue <= 0); 3 days falls in the 1-7 tier (3 * 0.50); 15 days falls in the 8-30 tier (3.50 + 8 * 1.00); 45 days exceeds 30 and is capped at 50.00. And isbnFromScanner, built with new String(...), is guaranteed to be a different object from the pooled literal isbnFromCatalog -- so what does == print, and what does .equals() print?

3.8 Compiling and Running Your Program

Nothing changes about the process: javac Book.java LibraryDemo.java, then java LibraryDemo. Java 14+ switch expressions require no special compiler flag on any JDK from 21 onward (the feature was finalized as a permanent language feature in Java 14, well before this course's JDK 21/25 baseline).

```
$ javac Book.java LibraryDemo.java
$ java LibraryDemo
```

3.9 OOP in Practice: Profitina

About this box

As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author -- describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's own release process is gated by an automated test suite of more than 978 tests (pytest) that must pass in full before any code change reaches the production server -- exactly the discipline Section 3.6's assert-vs-input distinction and bisection technique both serve, scaled from one student's program to a live product used by real businesses. A failing test in that suite is diagnosed with the same core tools this chapter introduces: inspecting a specific value at a specific point, narrowing down where an unexpected result first appears, and never trusting an assumption that hasn't actually been checked.

Profitina's backend also does substantial branching and tiered categorization internally -- for instance, deciding how to present a business's AI-visibility results involves comparing a computed value against a small number of thresholds, conceptually the same shape as this chapter's calculateFine tiers, just applied to a different problem. And because Profitina's AI-visibility engine parses text produced by external AI answer engines to look for brand mentions, it relies constantly and correctly on Python's own value-based string comparison (the Python track's Chapter 3 covers this directly) -- never on comparing string identity, which this chapter's Java Section 3.5 shows is a different, easily-confused question.

3.10 Chapter Summary and Key Takeaways

- Every value is either an object reference or one of Java's eight primitive types; int and double cover the overwhelming majority of real use.
- if/else if/else and switch both encode branching; prefer switch when one variable is compared against a small set of exact constants, and always prefer the modern arrow-form switch, which cannot fall through.
- for suits a known iteration count, while checks its condition before the first iteration (may run zero times), do-while checks after (always runs at least once).
- Strings are immutable and (for literals) pooled; == compares identity and .equals() compares content -- always use .equals() to compare String content.
- System.out debugging, IDE breakpoints, and assert each fit different situations; bisection localizes an unknown bug quickly; assert documents your OWN code's invariants and is never a substitute for validating untrusted input.
- Book gained calculateFine(int) and renewalStatusLabel() this chapter, adding no new fields -- both methods are pure functions of state Book already had.

Every tool this chapter introduced -- primitive types, branching, loops, String comparison, debugging technique -- existed already in every program you have written so far, used without being named. Naming them is what lets you reason about them deliberately instead of by accident.

3.11 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Lecture 4 (Quiz 1, covering Lectures 1-3).

1. Rewrite calculateFine's if/else if/else chain as a single expression using nested ternary operators (condition ? a : b). Is the result more or less readable than the if/else if/else version? Justify your answer.
2. Explain, precisely, why 0.1 + 0.2 == 0.3 is false in Java, and state the correct way to compare two double values for "close enough" equality.
3. What is printed by System.out.println("exit" == "exit")? What is printed by System.out.println("exit" == new String("exit"))? Explain the difference using Figure 3.4.
4. Write a short loop-based method countOverdueTiers(int[] daysArray) that returns how many books in the array fall into the 8-30 day fine tier. Which loop form (for, while, do-while, or enhanced for) fits best, and why?
5. Section 3.6 says assert should never validate untrusted input. Rewrite this line so it correctly rejects a negative daysOverdue argument to calculateFine even when assertions are disabled: assert daysOverdue >= 0;

Appendix 3.A — Execution Verification Log

The complete Book.java and LibraryDemo.java files (Code/Java/Chapter03/, provided alongside this book) were compiled and executed on OpenJDK 21.0.10 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ javac Book.java LibraryDemo.java
(no errors)

$ java LibraryDemo
"Clean Code" by Robert C. Martin [ISBN 978-0132350884, 2008] - on the shelf
Renewal status: Renewed once
Overdue  0 day(s) -> fine: 0.00
Overdue  3 day(s) -> fine: 1.50
Overdue 15 day(s) -> fine: 11.50
Overdue 45 day(s) -> fine: 50.00
== comparison:        false
.equals() comparison: true
```

References

[1] Oracle Corporation. "Primitive Data Types." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html (accessed August 2026).

[2] Oracle Corporation. "JEP 361: Switch Expressions." https://openjdk.org/jeps/361 (accessed August 2026) — the expression form of switch, a permanent language feature since Java 14.

[3] Oracle Corporation. "Java SE Support Roadmap." https://www.oracle.com/java/technologies/java-se-support-roadmap.html (accessed August 2026).

[4] R. C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008 (the source of this chapter's running example title).

CHAPTER 4 · EXERCISE CHAPTER

Practice Problems: Classes, Fields, Encapsulation & Language Fundamentals

No new material here — just seven problems designed to make sure Lectures 1 through 3 actually stuck, before Quiz 1.

Learning Objectives — after working through this chapter you will be able to:

(1) Explain, in your own words, what a class, an object, a field, and a constructor each are, and why encapsulation matters. (2) Write a class from a plain-language specification: choose the right fields, write a constructor that assigns all of them, and expose state only through methods. (3) Write methods that perform primitive arithmetic correctly, including a value that must be normalized into a fixed range. (4) Write a method that returns a brand-new instance of its own class rather than mutating an existing one. (5) Write two classes that collaborate — one class's method calling a method on an instance of the other.

4.1 Recap: Lectures 1–3 in Brief

Lecture 1 introduced the shift from procedural thinking to object-oriented thinking — bundling data and the behavior that operates on it into a single unit, a class, rather than passing raw data between free-standing functions. Lecture 2 gave that idea a precise anatomy: fields hold an object's state, a constructor initializes that state exactly once at creation, and encapsulation (private fields, public methods) is what keeps outside code from reaching in and corrupting that state directly. Lecture 3 then filled in Java's language fundamentals underneath all of that — primitive types, the arithmetic and normalization rules that govern them, control flow, Strings, and the basic debugging tools you reach for when a program's output doesn't match what you expected (Figure 4.1).

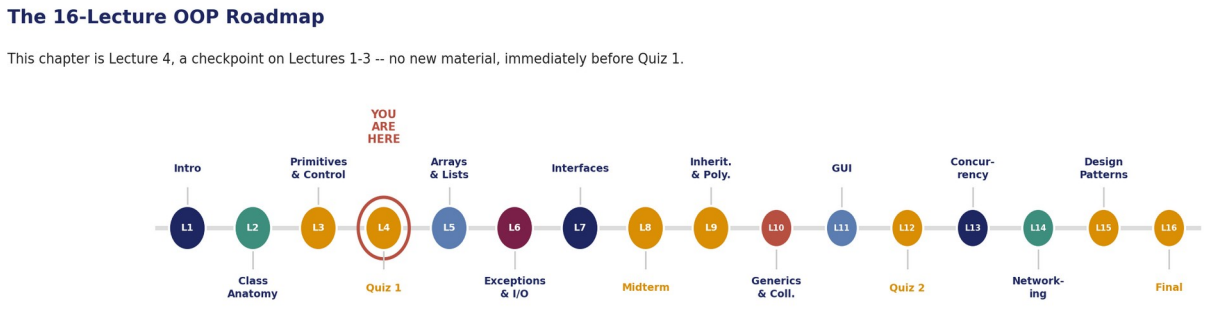


Figure 4.1 — This chapter sits at Lecture 4 in the sixteen-lecture OOP roadmap: a checkpoint, not a new topic, immediately before Quiz 1.

4.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a Bab Latihan: a practice chapter, not a new-content chapter. Every problem below is anchored to specific material from Lectures 1–3 (see Figure 4.2). Each problem comes with a HINT — a nudge toward the right way of thinking — but deliberately NOT a full worked solution or source code. Working through the problem yourself, even imperfectly, teaches far more than reading a finished answer.

Where Each Practice Problem Comes From

Every problem in Section 4.3 traces back to one of the previous three lectures.

#	Problem	Traces back to
1	Fraction Calculator	L3 -- primitive int arithmetic, static methods
2	Analog Clock Angle	L3 -- double arithmetic, modulo, normalization
3	Computer Account Class	L2 -- fields, constructor, encapsulation
4	Complex Number Class	L2 -- fields, methods returning new objects
5	Weight Conversion Class	L2 -- constructor, getters, a derived value
6	Advanced Fraction Class	L2/L3 -- toString(), methods returning the class's own type
7	Player and Enemy Classes	L2 -- two collaborating classes, mutation via a method

Figure 4.2 — Every problem in Section 4.3 traces back to one of the previous three lectures.

Before you start

Try each problem for real in a fresh .java project before reading its hint. If you get stuck, read only the hint — not a solution — and try again. This is exactly the discipline Section 4.4's Quiz 1 will reward: the quiz is open-book, but that is not a substitute for your own reasoning.

4.3 Practice Problems

4.3.1 Primitive Arithmetic & Control Flow

Problem 1 *[Easy]*. Let e1/d1 represent one fraction and e2/d2 a second fraction, where e1 and e2 are integers and d1 and d2 are positive integers. Write a class Q11Fraction that computes es/ds (the sum) and ep/dp (the product) of the two fractions, using the standard formulas: es/ds = (e1*d2 + e2*d1) / (d1*d2), and ep/dp = (e1*e2) / (d1*d2). Neither result needs to be reduced to lowest terms. Test your class on the pairs 1/2 & 1/3, 1/3 & 3/4, 1/2 & 2/3, and 1/4 & 2/3.

Hint

The problem statement already gives you both formulas directly — resist the urge to design a full Fraction object here (Problem 6 does that properly); the point of this one is translating es/ds and ep/dp into int arithmetic exactly as given, then testing all four pairs.

Problem 2 *[Easy]*. Write a class Q12Time with a static method angleBetween(int hours, int minutes) that computes the counterclockwise angle, in whole degrees normalized to 0-359, from the hour hand to the minute hand on a traditional analog clock. Recall: the minute hand moves 6 degrees per minute

(360/60), and the hour hand creeps forward 0.5 degree per minute in addition to 30 degrees per hour (360/12). Test at 9:00, 3:00, 18:00, 1:00, 2:30, and 4:41.

> **Hint**
>
> Compute the hour hand's angle including its 0.5-degree-per-minute creep, subtract the minute hand's angle, then normalize the result into 0-359 with a double modulo. Test the one example that is NOT a round number (4:41) before you trust your normalization — a rounding-vs-truncation bug can hide behind the round-number cases and only show up there.

4.3.2 Fields, Constructors & Encapsulation

Problem 3 *[Easy]*. Define a class Q13ComputerAccount, built from three Strings: realName, userName, and password. Implement printRealName(), printUserName(), and printPassword() (no arguments), plus changePassword(String newPassword) (no return value).

> **Hint**
>
> This is the plainest possible constructor-plus-accessor-methods class: three private fields, a constructor that assigns all three, one method per field to print it, and one method that mutates a single field. Nothing here should require any new concept beyond Lecture 2.

Problem 4 *[Medium]*. Define Q14ComplexNumber, storing a real part and an imaginary part, with a constructor and getters. Implement add, subtract, and multiply (each returning a new Q14ComplexNumber), and a toString() producing "a + bi". Use: (a+bi)+(c+di) = (a+c)+(b+d)i; (a+bi)-(c+di) = (a-c)+(b-d)i; (a+bi)*(c+di) = (ac-bd)+(ad+bc)i.

> **Hint**
>
> add, subtract, and multiply each take one Q14ComplexNumber argument and must return a BRAND-NEW Q14ComplexNumber, not modify either operand — the three arithmetic formulas are given to you directly in the problem statement, so the design work here is entirely about the return type, not the math.

Problem 5 *[Easy]*. Write Q15Weight(double pounds), storing a weight given in pounds, with getPounds() and getKilograms() (using 1 pound = 0.45359237 kilograms).

> **Hint**
>
> Store only the pounds value as a field. getKilograms() should compute the conversion on demand from that one field, rather than storing a second, redundant kilograms field that could fall out of sync if pounds were ever changed.

4.3.3 Bringing It Together

Problem 6 *[Medium]*. Define Q16Fraction with a constructor, getNumerator/getDenominator, a toString() (e.g. "1/2"), and getSum/getProduct, each returning a new Q16Fraction. For f1 = new Q16Fraction(1,2) and f2 = new Q16Fraction(3,7): f1.toString() should return "1/2"; f2.getProduct(f1) should print 3/14; f2.getSum(f1) should print 13/14.

> **Hint**
>
> This is Problem 1's two formulas again, but this time getSum and getProduct must each return a fully-formed Q16Fraction object, not a pair of raw ints — so the result can itself be summed or multiplied again without unpacking anything first.

Problem 7 *[Medium]*. Define two classes, Player and Enemy, each with name, health, power, and defense, plus a constructor, getters, and setters. Implement attack(opposing) (the opposing side takes damage equal to this side's power) and takeDamage(opponentAttack) (this side's health is reduced by opponentAttack - this.defense; if health drops below zero, print "{name} died!").

> **Hint**
>
> attack() should hand the opponent only ONE number — this side's power — and let the opponent's own takeDamage() apply its own defense and decide whether health has dropped below zero. Don't reach into the opponent's fields directly from inside attack(); the two classes should only ever talk to each other through their public methods.

4.4 Quiz 1 Format: Open-Book, Individual, Timed

Quiz 1 is an open-book, individual, timed take-home assessment (roughly 90 minutes) covering exactly the seven kinds of problems reviewed in this chapter — nothing beyond it. Grading is per-problem, not all-or-nothing: partial credit is given for a class that compiles and handles the general case correctly even if one edge case is missed.

Open-book means references, not collaborators

You may consult the textbook, your own notes, and lecture slides while answering. You may NOT collaborate with classmates or submit code that is not genuinely your own — an open-book quiz tests whether you can APPLY what you have studied, unsupervised.

Criterion	What it looks for	Weight
Correctness of output	Does the class produce the exact expected result for every given test case?	50%
Class design & encapsulation	Are fields private, and is state reachable only through the constructor and methods?	25%
Code clarity	Is the class easy to read, sensibly named, and free of dead or duplicated code?	15%
Compiles cleanly	Does it compile with javac with zero errors and zero warnings?	10%

4.5 OOP in Practice: Profitina

About this box

As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Every one of this chapter's seven problems follows the same shape that Profitina's own backend data model follows at far larger scale: private fields, a constructor that fully initializes them, and methods that are the only sanctioned way to read or change that state from outside the class. Before a change ships to production at Profitina, engineers run it through a short self-review checklist — very much like Appendix 4.A below — before ever requesting a second pair of eyes: does the change handle the exact inputs it was designed for, and has it been checked against edge cases a casual glance would miss? That habit of verifying your own reasoning before external review is exactly what makes an open-book take-home quiz format work, and it is precisely why Problem 2's 4:41 case and Problem 7's below-zero health check both matter far beyond this one practice chapter.

4.6 Chapter Summary

- This chapter introduced no new material — it is a checkpoint covering Lectures 1 through 3.
- Seven problems span the full range reviewed: primitive arithmetic and normalization (Lecture 3), and fields, constructors, encapsulation, and methods that return new objects (Lecture 2).
- Each problem includes a hint, not a solution — working through the reasoning and the code yourself is the point.
- Quiz 1 is an open-book, individual, timed take-home assessment: references are allowed, but the code must be your own, and correctness on every given test case is the majority of the grade.

A class that compiles is not the same as a class that is correct — the difference is always in the test cases you didn't think to try.

Appendix 4.A — Self-Check Checklist (Non-Graded)

Before submitting any answer — in this chapter's problems or in Quiz 1 itself — run it through this checklist. It costs a few minutes and catches most of the mistakes graders see most often.

- Does every class compile with javac with zero errors and zero warnings?
- Is every field private, with every value the caller needs exposed only through a method?
- Does each class's constructor assign every field it declares — no field left at its default value by mistake?
- Do Problems 4 and 6's arithmetic methods return a NEW object rather than mutating either operand?
- Did I test every example value the problem statement gives me, not just the ones that happen to be round numbers (Problem 2's 4:41 case exists specifically to catch a rounding bug that 9:00 and 3:00 cannot reveal)?
- Did I reread my own code as if I were a skeptical grader looking for the one input that breaks it?

References

[1] This exercise chapter is modeled on this course's real Quiz 1 (EF234302 Object-Oriented Programming, individual open-book take-home project) — same seven problems, same point weights.

[2] Oracle Corporation. "Classes." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/javaOO/classes.html (accessed August 2026).

CHAPTER 5

Arrays, Lists & the StringBuilder/StringBuffer Story

Every Book, Player, or Fraction object so far has existed alone. Real programs hold many of them at once -- a library's whole shelf, not one book. This chapter introduces Java's two containers for that job, the classic trap of building a long String piece by piece, and its fix.

Learning Objectives — after this chapter you will be able to:

(1) Declare, create, and index a fixed-size array, and state precisely why its length cannot change. (2) Declare an ArrayList<T>, and use add, remove, get, and size to manage a growing collection. (3) Explain why repeated String concatenation with + inside a loop is quadratic, and fix it with StringBuilder.append(). (4) State the one difference between StringBuilder and StringBuffer, and explain why StringBuilder is the correct default outside multi-threaded code. (5) Extend Pustaka with a Library class that manages a collection of Book objects, compile it, run it, and verify it yourself.

5.1 From One Book to a Shelf of Books

Chapters 1 through 4 worked with individual objects: one Book, one Fraction, one Player. Every real library, of course, holds many books at once, and a program that can only ever hold one Book object is not yet a library system at all -- it is a demonstration of a single Book. This chapter closes that gap by introducing Java's two ways to hold a collection of objects, and along the way meets a second, seemingly unrelated problem -- building up a long piece of text a little at a time -- that turns out to share the exact same underlying cost trap, and the exact same shape of fix (Figure 5.1).

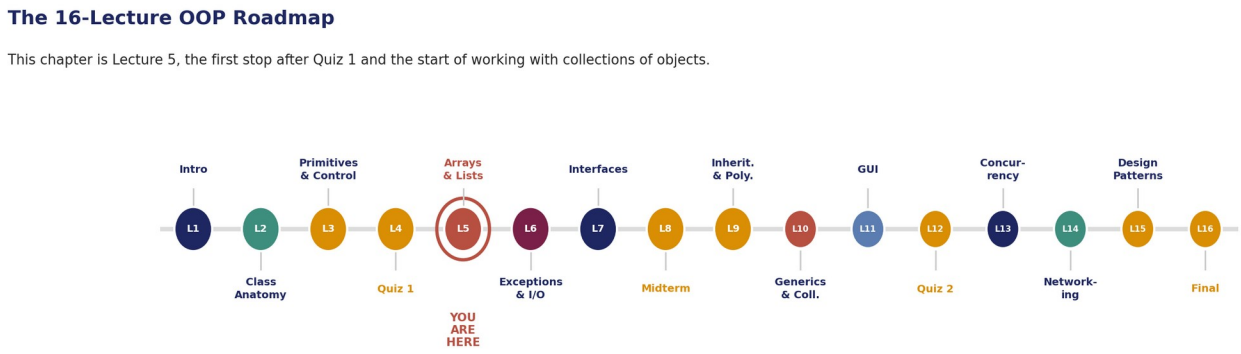


Figure 5.1 — The 16-lecture OOP roadmap. This chapter is Lecture 5, the first stop after Quiz 1 and the beginning of working with collections of objects.

5.2 Arrays: Java's Original, Fixed-Size Container

A Java array is declared with a type and square brackets, then created with new and a fixed length that can never change afterward: Book[] shelf = new Book[3]; creates room for exactly three Book references, indexed 0 through 2. Reading or writing shelf[3] on that array -- one past its length -- throws an ArrayIndexOutOfBoundsException at runtime; Java never silently allows it and never resizes the array to accommodate it.

Arrays are the lowest-level container Java offers: fast, simple, and directly supported by the language itself (that square-bracket syntax is not a class you call methods on). Their one serious limitation is exactly their fixed length -- useful when you know the exact count in advance (a chessboard is always 8x8), a poor fit when you do not, which describes almost every real collection of business objects, Pustaka's shelf of books included.

Fixed-Size Array vs. Growable List

Java's array and ArrayList, and Python's single built-in list, compared on the properties that matter day to day.

Property	Java array (Book[])	Java ArrayList<Book>	Python list
Size	`Fixed at creation`	`Grows/shrinks automatically`	`Grows/shrinks automatically`
Declare + create	`Book[] b = new Book[10];`	`List<Book> b = new ArrayList<>();`	`b = []`
Add an element	`b[i] = book;  (i must exist)`	`b.add(book);`	`b.append(book)`
Remove an element	`not directly supported`	`b.remove(book);`	`b.remove(book)`
Size right now	`b.length`	`b.size()`	`len(b)`
Element type	`Any type, incl. primitives`	`Objects only (no int; use Integer)`	`Any type, mixed if you want`

Java's array is the lowest-level container -- fast and simple, but its length is locked in forever.
ArrayList wraps a resizable array internally; Python's list is dynamic by default, with no separate fixed-size sibling to reach for.

Figure 5.2 — Java's array and ArrayList compared against Python's single built-in list. Notice ArrayList's size() and add() read almost identically to Python's len() and append().

5.3 ArrayList<T>: A Collection That Grows With You

java.util.ArrayList<T> solves exactly the problem Section 5.2 raised: List<Book> shelf = new ArrayList<>(); starts empty and grows automatically every time shelf.add(book) is called -- there is no length to declare up front and no ceiling to hit. The <Book> in List<Book> is a generic type parameter (covered fully in Chapter 10), read for now simply as "this list holds Book objects, and only Book objects" -- attempting shelf.add("not a book") is a compile error, not a runtime surprise.

ArrayList's core operations, at a glance

add(item) appends to the end; get(i) reads the item at index i; remove(item) or remove(i) removes by value or by index; size() reports the current count (there is no .length -- that only exists on true arrays). An enhanced for-loop (for (Book b : shelf)) iterates an ArrayList exactly the way it iterates a plain array -- one more reason ArrayList is almost always the right default choice over a raw array for a collection of objects.

ArrayList<int> does not compile

Java generics work only with object types, never with primitives -- List<int> is a compile error. Use the boxed wrapper type instead: List<Integer>, and Java will automatically convert ("autobox") between int and Integer as needed. This is the one situation Figure 5.2's "objects only" row is warning about.

5.4 The StringBuilder/StringBuffer Story

Strings, as Chapter 3 established, are immutable: every += on a String does not modify it in place but creates an entirely new String object and copies the old content into it. Inside a loop that runs n times, that means the first iteration copies a tiny string, the second copies a slightly longer one, and so on -- total copying work grows proportional to n^2 (quadratic), not n, even though the loop itself only runs n times (Figure 5.3).

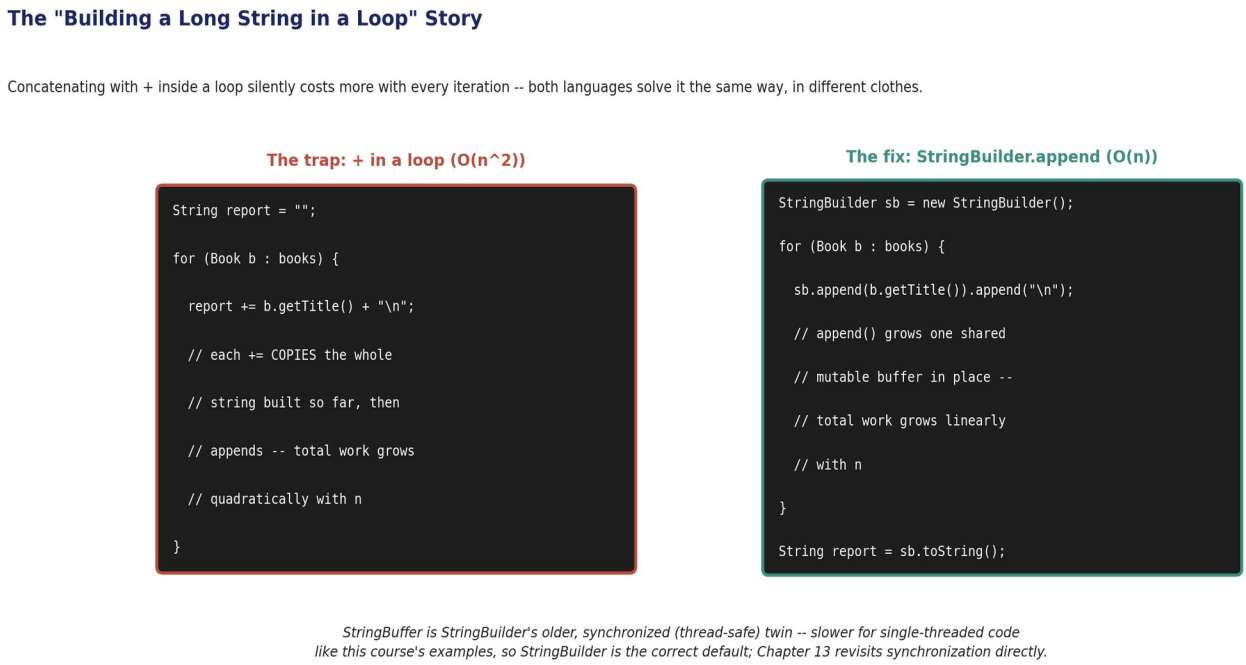


Figure 5.3 — The same report-building loop written the costly way (String += in a loop) and the correct way (StringBuilder.append() in a loop, converted to a String only once at the end).

StringBuilder solves this by being genuinely mutable: append() grows one shared internal buffer in place, so total work across the whole loop is proportional to n, not n^2. Only after every piece has been appended is .toString() called once, producing the final String. This is precisely the pattern Library.generateReport() uses in Section 5.5 below.

> **StringBuilder vs. StringBuffer**
>
> StringBuffer is functionally identical to StringBuilder -- same methods, same behavior -- except every method is synchronized, meaning only one thread may call it at a time. That synchronization costs real performance and buys nothing in ordinary single-threaded code, which describes every example in this course so far. StringBuilder is therefore the correct default; reach for StringBuffer only when multiple threads genuinely share one builder, a scenario Chapter 13 (Concurrency) examines directly.

5.5 Extending Pustaka: The Library Class

Library, introduced this chapter, manages a List<Book> and puts Sections 5.3 and 5.4 to work together: addBook and removeBook manage the collection, and generateReport builds a multi-line summary of the whole shelf using StringBuilder rather than +=.

Phase 5 — Implement (Library.java, abridged)

```
public class Library {
    private List<Book> books = new ArrayList<>();

    public void addBook(Book book) { books.add(book); }

    public boolean removeBook(String isbn) {
        for (int i = 0; i < books.size(); i++) {
            if (books.get(i).getIsbn().equals(isbn)) {
                books.remove(i);
                return true;
            }
        }
        return false;
    }

    public String generateReport() {
        StringBuilder sb = new StringBuilder();
        sb.append("Pustaka Library Report (");
        sb.append(books.size()).append(" book(s))\n");
        for (Book b : books) {
            sb.append("- ").append(b.getTitle());
            sb.append(" by ").append(b.getAuthor());
            sb.append(b.isOnLoan() ? " [on loan]" : " [on shelf]").append("\n");
        }
        return sb.toString();
    }
}
```

Notice removeBook uses a plain indexed for-loop rather than a lambda-based ArrayList.removeIf -- that shorter form exists in modern Java, but relies on functional interfaces this course has not covered yet (Chapter 7 and beyond); a plain loop over get(i) is entirely correct here and uses only tools already introduced.

5.6 Compiling and Running Your Program

Nothing changes about the process: javac Book.java Library.java LibraryDemo.java, then java LibraryDemo. java.util.ArrayList and java.util.List need one import line each at the top of any file that uses them directly (Library.java has both; LibraryDemo.java uses Library only, so it needs neither import itself).

```
$ javac Book.java Library.java LibraryDemo.java
$ java LibraryDemo
```

5.7 OOP in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author -- describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's backend routinely holds collections whose size is never known in advance -- a business's set of tracked keywords, a batch of AI-answer-engine responses fetched for one analysis run -- exactly the situation Section 5.2 argues a fixed-size array cannot serve well. Every one of those collections is backed by a dynamically-sized structure, Python's own list on the backend (the Python track's Chapter 5 covers this side directly) conceptually mirroring Java's ArrayList here. And because Profitina's reporting layer regularly assembles longer text -- a formatted summary of a business's AI-visibility results, for instance -- it follows the same StringBuilder-shaped discipline Section 5.4 introduces: build the pieces first, join them once, never concatenate in a loop.

5.8 Chapter Summary and Key Takeaways

- A Java array has a fixed length set at creation and never changes; ArrayList<T> grows and shrinks automatically and is almost always the better default for a collection of objects.
- ArrayList's core operations are add, remove, get, and size -- no .length, which is array-only syntax.
- Generics (the <Book> in List<Book>) work only with object types; use the boxed wrapper (Integer, not int) for a list of numbers.
- String += inside a loop is quadratic (O(n^2)) because every += copies the whole string built so far; StringBuilder.append() is linear (O(n)) because it grows one shared mutable buffer in place.
- StringBuffer is StringBuilder's synchronized twin -- correct only when multiple threads share one builder; StringBuilder is the correct single-threaded default.
- Library gained addBook, removeBook, findByTitle, size, and generateReport this chapter -- Book itself did not change at all.

Every collection you will manage for the rest of this course -- a library's shelf, a roster of players, a batch of network requests -- builds on exactly the two ideas this chapter introduced: a container that grows with you, and text assembled once instead of copied over and over.

5.9 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Why does Book[] shelf = new Book[5]; followed later by shelf[5] = someBook; throw an exception, while shelf2.add(someBook) on an ArrayList<Book> of size 5 never does?
2. Rewrite Library.generateReport() using String += instead of StringBuilder. For a library of 10,000 books, roughly how many times more total character-copying work would this version do compared to the StringBuilder version? (Hint: compare n vs. n^2 for n = 10,000.)

3. Why is List<int> a compile error in Java, and what is the correct way to declare a list of whole numbers?
4. Add a method Library.countOnLoan() that returns how many books in the collection are currently on loan, using an enhanced for-loop and Book's existing isOnLoan() getter.
5. In your own words, explain why StringBuffer would be the correct choice instead of StringBuilder if two separate threads were both appending to the same report at the same time. (You are not expected to write thread-safe code yet -- Chapter 13 covers that. This question checks that you understand WHY the two classes differ, not how to use threads.)

Appendix 5.A — Execution Verification Log

The complete Book.java, Library.java, and LibraryDemo.java files (Code/Java/Chapter05/, provided alongside this book) were compiled and executed on OpenJDK 21.0.10 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ javac Book.java Library.java LibraryDemo.java
(no errors)

$ java LibraryDemo
Fixed shelf (array), length = 2:
- Clean Code
- Effective Java

Library size after adding 3 books: 3
Pustaka Library Report (3 book(s))
- Clean Code by Robert C. Martin [on shelf]
- Effective Java by Joshua Bloch [on loan]
- The Pragmatic Programmer by Andrew Hunt [on shelf]
Removed The Pragmatic Programmer: true
Library size now: 2
Pustaka Library Report (2 book(s))
- Clean Code by Robert C. Martin [on shelf]
- Effective Java by Joshua Bloch [on loan]
```

References

[1] Oracle Corporation. "Arrays." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html (accessed August 2026).

[2] Oracle Corporation. "The ArrayList Class." The Java Tutorials. https://docs.oracle.com/javase/tutorial/collections/implementations/list.html (accessed August 2026).

[3] Oracle Corporation. "Class StringBuilder" / "Class StringBuffer." Java SE 21 API Documentation. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/StringBuilder.html (accessed August 2026).

[4] J. Bloch. Effective Java, 3rd ed. Addison-Wesley, 2017 (a second running title alongside "Clean Code" for this chapter's multi-book examples).

CHAPTER 6

Exception Handling & File I/O

Every method so far has assumed the happy path: valid input, a file that exists, an ISBN that's really in the library. Real programs cannot make that assumption. This chapter introduces Java's exception mechanism for handling failure gracefully, and file I/O for making Pustaka's data outlive the program that created it.

Learning Objectives — after this chapter you will be able to:

(1) Explain the difference between a checked and an unchecked exception, and when Java forces you to declare or catch one. (2) Write a try-catch-finally block, and state precisely when the finally block runs. (3) Define a custom checked exception by extending Exception, and throw and catch it correctly. (4) Read and write a plain text file using try-with-resources, and explain why try-with-resources is preferred over a manual try-finally with close(). (5) Extend Pustaka with save/load persistence and a stricter, exception-throwing removal method, compile it, run it, and verify it yourself.

6.1 From In-Memory Objects to Programs That Can Fail

Chapters 1 through 5 built a Pustaka that works perfectly -- as long as nothing ever goes wrong. removeBook silently returns false if the ISBN is not found, and every book the program has ever seen lives only in memory: close the program and the whole library vanishes. Neither is acceptable in a real system. This chapter fixes both gaps: Java's exception mechanism gives methods a principled way to signal "this specific thing failed," and file I/O gives Library a way to persist its books to disk and reload them later (Figure 6.1).

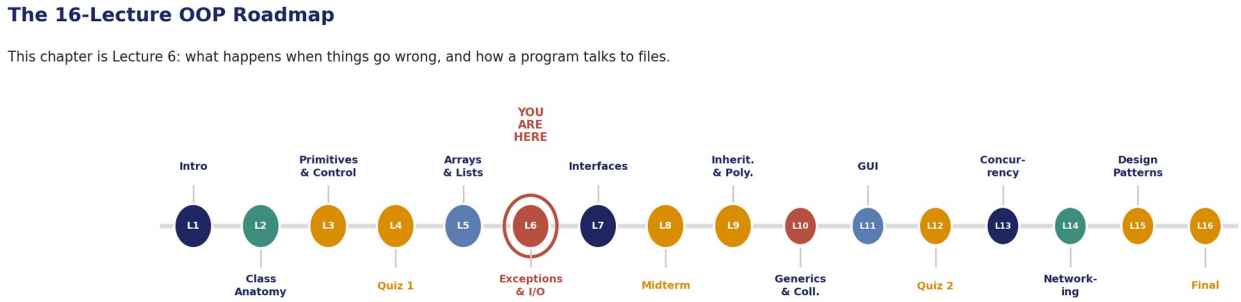


Figure 6.1 — The 16-lecture OOP roadmap. This chapter is Lecture 6: what happens when things go wrong, and how a program talks to files.

6.2 Java's Exception Hierarchy: Checked vs. Unchecked

Every error condition in Java is represented by an object -- ultimately, every exception class descends from Throwable. Error and its subclasses represent JVM-level failures (running out of memory, a stack overflow) that ordinary code is not expected to catch. Exception and its subclasses represent conditions a program can reasonably be expected to handle. Within Exception, Java draws one more crucial line: RuntimeException and its subclasses (NullPointerException, ArrayIndexOutOfBoundsException, and similar) are UNCHECKED -- the compiler never forces you to

catch or declare them, because they usually indicate a programming bug rather than an expected failure mode. Every other Exception subclass is CHECKED: the compiler forces every caller to either catch it or declare it with throws, so a checked exception can never be silently forgotten about (Figure 6.2).

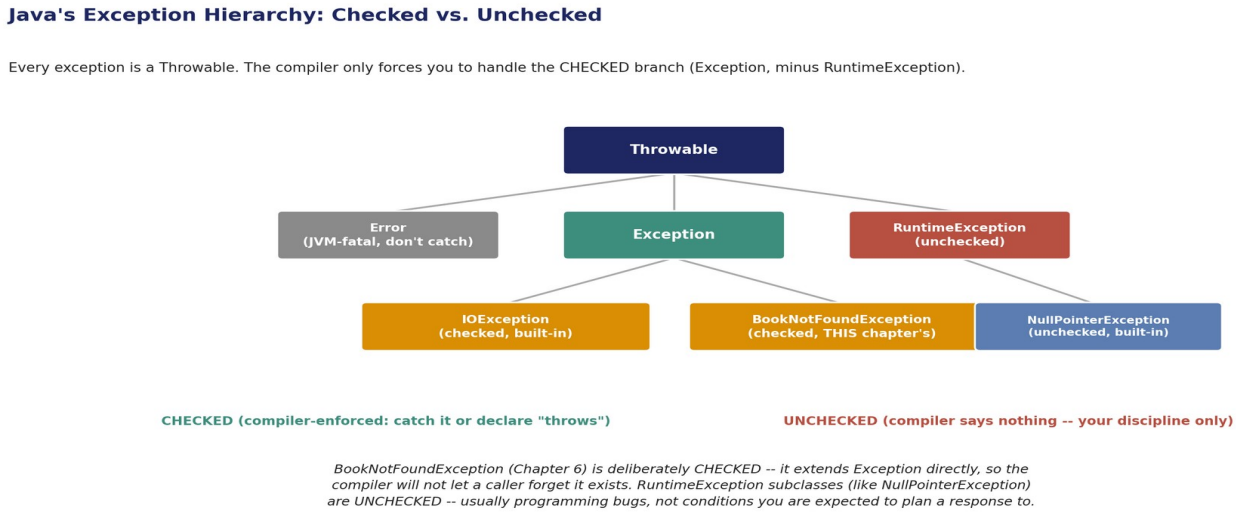


Figure 6.2 — Java's exception hierarchy. This chapter's own BookNotFoundException is deliberately checked, sitting alongside the built-in checked IOException.

> **Why make BookNotFoundException checked at all?**
>
> A checked exception is the right tool exactly when the failure is a normal, expected outcome the caller genuinely needs to plan a response to -- "the ISBN you asked to remove isn't in this library" is exactly that kind of condition, not a bug. Making it checked means the compiler itself will not let a future maintainer of this code accidentally call removeBookOrThrow and forget that it can fail.

6.3 try, catch, and finally

A try block wraps code that might throw; one or more catch blocks each handle one specific exception type; an optional finally block runs no matter what -- whether the try block completed normally, threw an exception that was caught, or threw one that was not caught and is about to propagate further up the call stack (Figure 6.3).

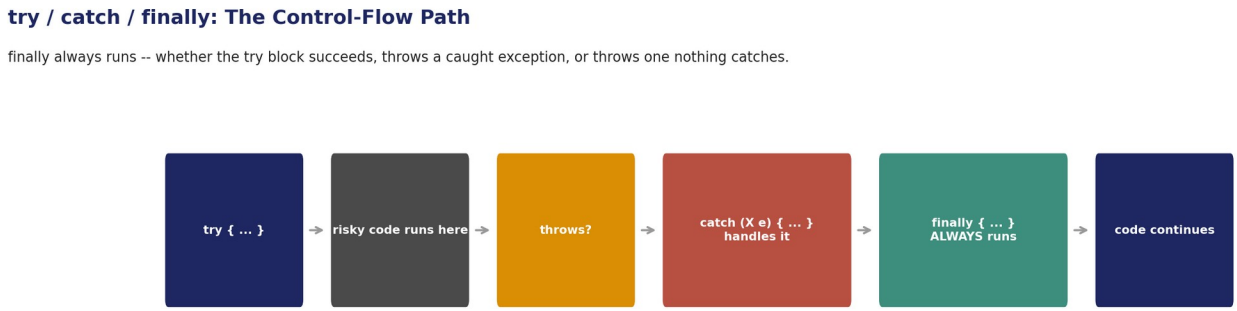


Figure 6.3 — The try/catch/finally control-flow path. finally's defining property is that it always executes.

A minimal try-catch-finally

```
try {
    library.removeBookOrThrow(isbn);
} catch (BookNotFoundException e) {
    System.out.println("Caught expected exception: " + e.getMessage());
} finally {
    System.out.println("This always prints, exception or not.");
}
```

Catching Exception (or Throwable) is almost always wrong

catch (Exception e) { } swallows every checked exception indiscriminately -- including ones you never anticipated and have no correct way to handle -- and an empty catch block silently discards the failure entirely, which is far worse than letting the program crash with a clear stack trace. Always catch the most SPECIFIC exception type your code can actually do something meaningful about.

6.4 Custom Exceptions: BookNotFoundException

Defining a custom exception is ordinary inheritance: extend Exception (for a checked exception, this chapter's choice) or RuntimeException (for an unchecked one), call super(message) in the constructor, and the new class is a fully legitimate exception type Java will let you throw and catch by name.

Phase 6 — Implement (BookNotFoundException.java)

```
public class BookNotFoundException extends Exception {
    private final String isbn;

    public BookNotFoundException(String isbn) {
        super("No book found with ISBN " + isbn);
        this.isbn = isbn;
    }

    public String getIsbn() { return isbn; }
}
```

Library.removeBookOrThrow (Section 6.6) throws this exception when removeBook's ISBN lookup fails, giving callers who need to know about that failure a way to be certain they cannot accidentally ignore it -- the compiler enforces it.

6.5 File I/O with try-with-resources

Reading or writing a file opens a system resource (a file handle) that MUST be closed when you are done with it, or the resource leaks -- a real problem in a long-running program that opens many files over its lifetime. Java 7 introduced try-with-resources specifically to make this safe by default: any object declared inside the try(...) parentheses that implements AutoCloseable has its close() method called automatically when the try block exits, whether normally or via an exception -- no explicit finally block needed.

try-with-resources vs. the old manual pattern

```
// Modern (Java 7+): try-with-resources -- always prefer this
try (BufferedReader reader = new BufferedReader(new FileReader(path))) {
    String line = reader.readLine();
    // reader.close() is called automatically here, guaranteed
}

// Old manual pattern -- verbose and easy to get wrong
BufferedReader reader = new BufferedReader(new FileReader(path));
try {
    String line = reader.readLine();
} finally {
    reader.close();  // easy to forget, or to place incorrectly
}
```

IOException is checked, on purpose

Every method that touches the filesystem can fail for reasons entirely outside your program's control -- a missing file, a full disk, a permissions error -- so java.io.IOException is a checked exception, and Java forces every method that reads or writes a file to either catch IOException or declare throws IOException. Section 6.6's saveToFile and loadFromFile both declare it, passing the responsibility for deciding how to react up to their own caller.

6.6 Extending Pustaka: Persistence and Stricter Removal

Library gains two related capabilities this chapter: removeBookOrThrow, a stricter sibling of removeBook (Chapter 5) that throws BookNotFoundException instead of quietly returning false; and saveToFile/loadFromFile, simple text-file persistence built on try-with-resources.

Phase 6 — Implement (Library.java, abridged)

```
public void removeBookOrThrow(String isbn) throws BookNotFoundException {
    if (!removeBook(isbn)) {
        throw new BookNotFoundException(isbn);
    }
}

public void saveToFile(String path) throws IOException {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(path))) {
        for (Book b : books) {
            writer.write(b.getTitle() + "|" + b.getAuthor() + "|"
                + b.getIsbn() + "|" + b.getPublicationYear());
            writer.newLine();
        }
    }
}

public void loadFromFile(String path) throws IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(path))) {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split("\\|", -1);
            books.add(new Book(parts[0], parts[1], parts[2],
                Integer.parseInt(parts[3])));
        }
    }
}
```

Each line written by saveToFile is a plain pipe-delimited record -- title|author|isbn|year -- deliberately simple rather than a real serialization format (Chapter 14's networking work and beyond will introduce JSON), because the point of this chapter is the exception-handling and resource-management discipline, not the file format itself.

6.7 Compiling and Running Your Program

Nothing changes about the process: javac BookNotFoundException.java Book.java Library.java LibraryDemo.java, then java LibraryDemo. java.io.BufferedReader, BufferedWriter, FileReader, FileWriter, and IOException each need an import line at the top of Library.java; LibraryDemo.java needs only java.io.IOException, since it calls Library's methods rather than the I/O classes directly.

```
$ javac BookNotFoundException.java Book.java Library.java LibraryDemo.java
$ java LibraryDemo
```

6.8 OOP in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author -- describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's backend calls external AI answer engines and third-party APIs constantly -- calls that can time out, return malformed data, or fail outright for reasons entirely outside Profitina's own control, precisely the situation Section 6.2 describes checked exceptions existing for. Every such call is wrapped in its own try-catch with a specific, named exception type -- never a bare catch (Exception e), for exactly the reason Section 6.3's trap callout warns against -- so a genuine failure is always visible and handled deliberately, never silently swallowed. And because Profitina's backend persists data far more durably than this chapter's plain text file (a real database, not a text file -- Chapter 14 and the Databases/FBP course cover that machinery directly), the underlying discipline is identical: never leave a resource open longer than it needs to be, and never let an I/O failure pass unnoticed.

6.9 Chapter Summary and Key Takeaways

- Every Java exception descends from Throwable; RuntimeException and its subclasses are UNCHECKED (compiler-optional), every other Exception subclass is CHECKED (compiler-enforced).
- try-catch-finally: finally always runs, whether the try block succeeds, throws a caught exception, or throws one that propagates uncaught.
- A custom exception is ordinary inheritance -- extend Exception for checked, RuntimeException for unchecked -- and BookNotFoundException (this chapter's) is deliberately checked.
- Always catch the most specific exception type possible; never leave a catch block empty, and avoid catching bare Exception.
- try-with-resources automatically closes any AutoCloseable resource (a file reader/writer here) when the try block exits, replacing the old, error-prone manual try-finally-close() pattern.
- Library gained removeBookOrThrow, saveToFile, and loadFromFile this chapter -- Book itself did not change at all.

A program that only ever runs on perfect input, against a file that always exists, is not yet a real program -- it is a demonstration of the happy path. Exception handling and file I/O are what let Pustaka survive contact with the real world.

6.10 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Is NullPointerException checked or unchecked? Explain how you can tell from the class hierarchy alone, without looking it up.
2. Rewrite the try-with-resources example in Section 6.5 as the old manual try-finally-close() pattern, and identify one concrete way a maintainer could get the manual version subtly wrong that try-with-resources makes impossible.
3. Why does Library.saveToFile declare throws IOException instead of catching it internally and printing an error message?
4. Add a method Library.countByAuthor(String author) that returns how many books in the collection were written by that author, and decide (with a one-sentence justification) whether a missing author should throw an exception or simply return 0.

5. In your own words, explain why an empty catch (Exception e) { } block is considered one of the worst mistakes a Java program can make, even though it compiles and runs without any visible error.

Appendix 6.A — Execution Verification Log

The complete BookNotFoundException.java, Book.java, Library.java, and LibraryDemo.java files (Code/Java/Chapter06/, provided alongside this book) were compiled and executed on OpenJDK 21.0.10 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ javac BookNotFoundException.java Book.java Library.java LibraryDemo.java
(no errors)

$ java LibraryDemo
Library size after adding 3 books: 3
Caught expected exception: No book found with ISBN 0000000000000
Library size now: 2
Pustaka Library Report (2 book(s))
- Clean Code by Robert C. Martin [on shelf]
- Effective Java by Joshua Bloch [on shelf]
Saved library to library_export.txt
Reloaded library size: 2
Pustaka Library Report (2 book(s))
- Clean Code by Robert C. Martin [on shelf]
- Effective Java by Joshua Bloch [on shelf]
```

References

[1] Oracle Corporation. "Exceptions." The Java Tutorials. https://docs.oracle.com/javase/tutorial/essential/exceptions/ (accessed August 2026).

[2] Oracle Corporation. "The try-with-resources Statement." The Java Tutorials. https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html (accessed August 2026).

[3] Oracle Corporation. "Basic I/O." The Java Tutorials. https://docs.oracle.com/javase/tutorial/essential/io/ (accessed August 2026).

[4] J. Bloch. Effective Java, 3rd ed. Addison-Wesley, 2017, Chapter 10 ("Exceptions").

CHAPTER 7

Interfaces & Abstract Classes

So far, Pustaka has had exactly one kind of thing in it: a Book. Real libraries lend DVDs, magazines, and equipment too, and none of that content is a book at all. This chapter introduces two tools Java gives you for sharing structure and behavior across genuinely different classes -- abstract classes, for types that share real state and code, and interfaces, for capabilities that cut across any single hierarchy.

Learning Objectives — after this chapter you will be able to:

(1) Explain why an abstract class cannot be instantiated, and write one with both abstract and concrete methods. (2) Define a Java interface, including a default method, and explain how implementing multiple interfaces differs from extending one class. (3) State a working rule of thumb for choosing an abstract class versus an interface for a given design problem. (4) Refactor an existing class to extend an abstract superclass and implement an interface, without breaking any of its existing behavior. (5) Write a loop that calls a method polymorphically across two unrelated concrete subtypes, compile it, run it, and verify the output yourself.

7.1 Recap: Pustaka So Far

Chapters 1 through 6 built a single, ever-more-capable Book class and a Library that manages a collection of them: fields and encapsulation (Chapter 2), primitive types and control flow (Chapter 3), arrays and Lists (Chapter 5), and exception handling with file persistence (Chapter 6). Every one of those chapters improved Book or Library without ever asking a harder question: what happens when Pustaka needs to hold something that is not a Book at all (Figure 7.1)?

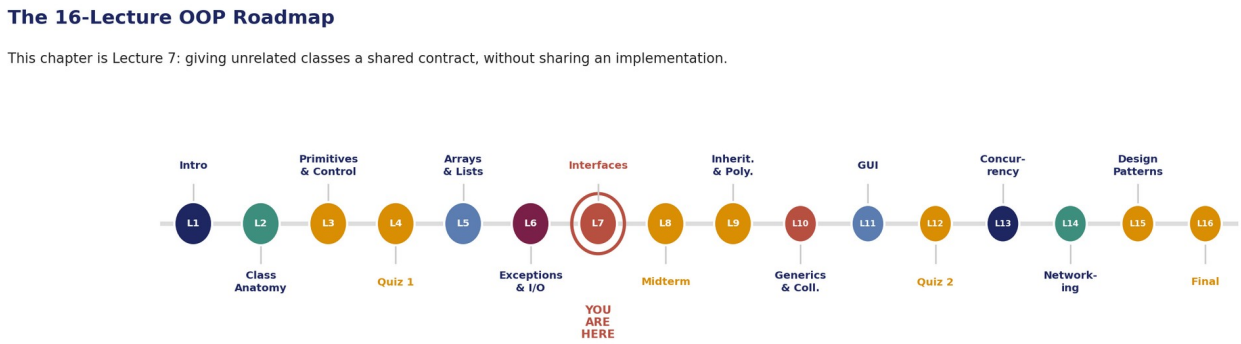


Figure 7.1 — The 16-lecture OOP roadmap. This chapter is Lecture 7: giving unrelated classes a shared contract, without sharing an implementation.

7.2 Abstract Classes: Shared State, Shared Code, No Instances

An abstract class is declared with the abstract keyword and may contain both abstract methods (declared but not implemented -- every concrete subclass MUST override them) and ordinary concrete methods (implemented once, inherited unchanged by every subclass). Java will not let you write new on an abstract class directly -- only a concrete subclass that has implemented every abstract method can be instantiated. This is the tool of choice when subtypes genuinely share fields and real, working code, not just a naming convention.

A minimal abstract class

```
public abstract class LibraryItem {
    protected final String title;
    protected boolean onLoan;

    // Concrete method -- every subclass inherits this exact code.
    public boolean isOnLoan() { return onLoan; }

    // Abstract method -- every subclass MUST supply its own version.
    public abstract double calculateFine(int daysOverdue);
}
```

Java allows only single inheritance

A class may extend exactly one superclass, abstract or not -- LibraryItem's subclasses in this chapter (Book, DVD) cannot also extend some second class. This is precisely the gap interfaces exist to fill: a class can implement as many interfaces as it needs, alongside its one superclass.

7.3 Interfaces: A Capability, Not a Rung on a Ladder

An interface declares a set of methods a class promises to provide, with no state (fields) of its own and, traditionally, no implementation at all -- just a contract. A class opts in with implements, and unlike extends, a class may implement as many interfaces as it needs. Since Java 8, an interface may also include default methods: a full method body every implementing class inherits for free, unless it chooses to override it.

A minimal interface with a default method

```
public interface Renewable {
    boolean renew();
    String renewalStatusLabel();

    // Default method (Java 8+): built entirely out of the two
    // methods above -- every implementing class gets this for
    // free, and may still override it if it needs to.
    default String renewalReceipt(boolean ok) {
        return "Renewal attempt: " + (ok ? "SUCCESS" : "DENIED")
            + " -- " + renewalStatusLabel();
    }
}
```

Notice what Renewable does NOT mention: it says nothing about titles, ISBNs, or whether an item is currently on loan. It only describes one narrow capability -- can this thing's loan be extended? -- completely independent of what kind of thing it is. A Book and a DVD can both implement Renewable despite having almost nothing else in common, and a future item type that is never renewable (a reference-only encyclopedia, say) simply would not implement it at all (Figure 7.2).

7.4 Choosing Between an Abstract Class and an Interface

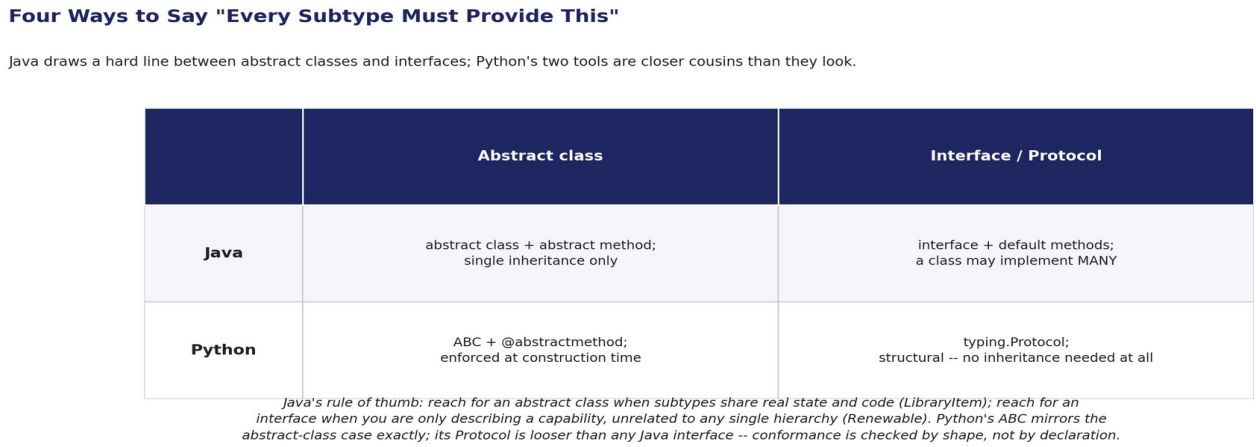


Figure 7.2 — Four ways to express "every subtype must provide this," across Java and Python.

A practical rule of thumb: reach for an abstract class when subtypes share real, non-trivial state and behavior that would otherwise be duplicated (LibraryItem's title, isbn, and onLoan fields, plus its concrete borrow() logic). Reach for an interface when you are describing a capability that cuts across any single hierarchy, and multiple unrelated classes might want to opt into it (Renewable). Many well-designed classes do both at once -- exactly what Book and DVD do in this chapter, extending LibraryItem AND implementing Renewable.

7.5 Python's Answer: ABC and Protocol

Python has no interface or abstract class keyword. The abc module's ABC base class plus the @abstractmethod decorator is the closest match to Java's abstract classes: a class inheriting from ABC that declares an @abstractmethod cannot be instantiated, and Python raises a TypeError at construction time if a concrete subclass forgets to override one -- one of the very few places Python enforces a contract at all, rather than trusting the caller the way exception handling does (Chapter 6).

For interface-like behavior, Python's typing.Protocol (PEP 544) takes a completely different, STRUCTURAL approach: a class satisfies a Protocol simply by having methods with matching names and signatures -- no inheritance declaration of any kind is required. Book and DVD in the Python track satisfy Renewable purely by having a renew() and a renewal_status_label() method; nothing ever writes class Book(Renewable). This is Python's duck-typing philosophy applied formally: "if it renews like a Renewable, it is one (Figure 7.3)."

7.6 Extending Pustaka: LibraryItem, Renewable, and a New DVD Type

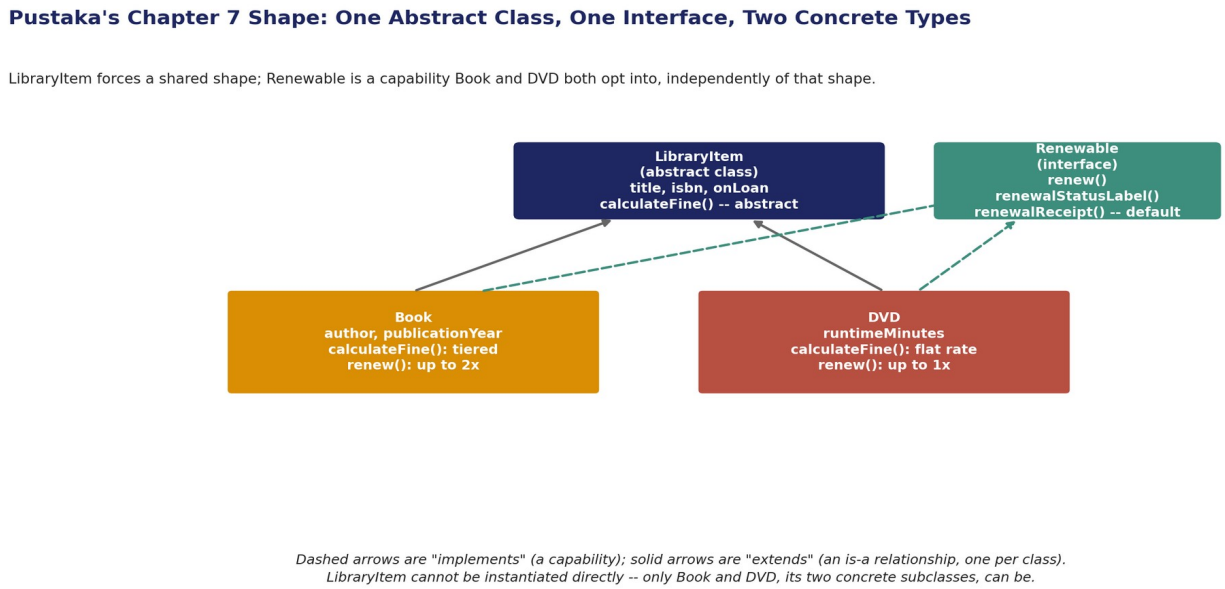


Figure 7.3 — Pustaka's Chapter 7 shape: LibraryItem is an abstract superclass; Renewable is an interface Book and DVD each opt into.

Book's title, isbn, and onLoan fields, along with its borrow()/returnItem() logic, move up into a new abstract class, LibraryItem, which also declares calculateFine() as abstract -- every concrete item type must supply its own overdue-fine rule, because a book's late policy and a DVD's late policy are genuinely different rules, not just different numbers. Book keeps only what is genuinely book-specific: author, publicationYear, and timesRenewed.

Phase 7 — Implement (LibraryItem.java, abridged)

```
public abstract class LibraryItem {
    protected final String title;
    protected final String isbn;
    protected boolean onLoan;

    protected LibraryItem(String title, String isbn) {
        this.title = title;
        this.isbn = isbn;
        this.onLoan = false;
    }

    public boolean borrow() {
        if (onLoan) { return false; }
        onLoan = true;
        return true;
    }

    public abstract double calculateFine(int daysOverdue);

    public String describe() {
        return String.format("\"%s\" [ISBN %s] - %s",
            title, isbn, onLoan ? "ON LOAN" : "on the shelf");
    }
}
```

DVD is a brand-new item type this chapter -- it shares nothing with Book except LibraryItem's fields and Renewable's contract. It renews at most once (Book allows two renewals) and charges a flat, steeper daily rate with no cap (Book's fine tiers off after 30 days).

Phase 7 — Implement (DVD.java, abridged)

```
public class DVD extends LibraryItem implements Renewable {
    private final int runtimeMinutes;
    private int timesRenewed;
    private static final int MAX_RENEWALS = 1;
    private static final double DAILY_FINE = 0.75;

    public DVD(String title, String isbn, int runtimeMinutes) {
        super(title, isbn);
        this.runtimeMinutes = runtimeMinutes;
    }

    @Override
    public double calculateFine(int daysOverdue) {
        return daysOverdue <= 0 ? 0.0 : daysOverdue * DAILY_FINE;
    }

    @Override
    public boolean renew() {
        if (!onLoan || timesRenewed >= MAX_RENEWALS) { return false; }
        timesRenewed++;
        return true;
    }

    @Override
    public String renewalStatusLabel() {
        return timesRenewed >= MAX_RENEWALS ? "No renewals left" : "Renewable
once";
    }
}
```

Book's borrow() overrides a CONCRETE inherited method

Book overrides LibraryItem's own borrow() -- not to redo the "already on loan?" check, but to add one book-specific step (resetting timesRenewed) on top of it, by calling super.borrow() first. Overriding a concrete method to extend it, rather than replace it outright, is just as legitimate as overriding an abstract one.

7.7 Polymorphism in Action: The Driver Program

The real payoff of this chapter is a single loop over a List<LibraryItem> containing both a Book and a DVD, where calculateFine(), renew(), and describe() are all called polymorphically -- the loop never once asks "is this a Book or a DVD?" Java dispatches each call to the correct override automatically, based on the object's actual runtime type (Figure 7.4).

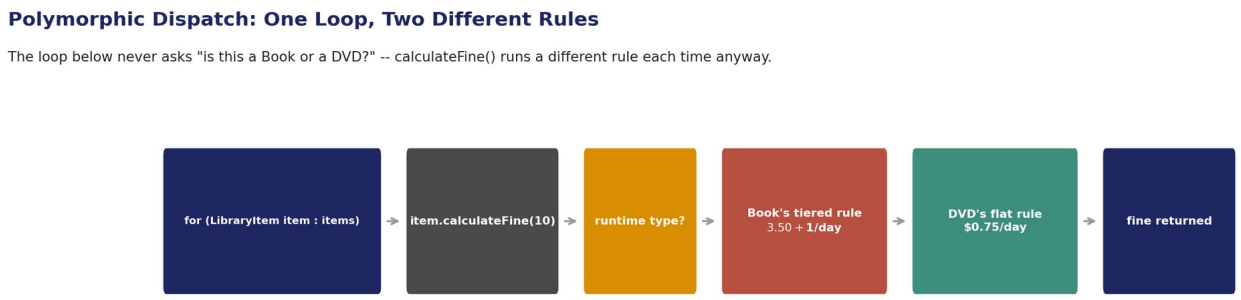


Figure 7.4 — Polymorphic dispatch: one loop body, two genuinely different calculateFine() rules.

Phase 7 — Implement (LibraryDemo.java, abridged)

```
List<LibraryItem> items = new ArrayList<>();
items.add(new Book("Clean Code", "Robert C. Martin", "9780132350884", 2008));
items.add(new DVD("The Imitation Game", "9900000000001", 114));

for (LibraryItem item : items) {
    item.borrow();
}

for (LibraryItem item : items) {
    System.out.printf("%s -> fine: $%.2f%n", item.describe(),
item.calculateFine(10));
}

for (LibraryItem item : items) {
    if (item instanceof Renewable r) {
        boolean ok = r.renew();
        System.out.println(item.getTitle() + ": " + r.renewalReceipt(ok));
    }
}
```

Library itself is deliberately left unchanged this chapter -- it still manages a List<Book> exactly as it has since Chapter 5. Generalizing Library to hold any LibraryItem is Chapter 9's job, once inheritance and polymorphism get a full chapter of their own; this chapter's driver program demonstrates the polymorphism directly, in a standalone list, so the idea is not tangled up with Library's file-persistence logic from Chapter 6.

7.8 Compiling and Running Your Program

javac LibraryItem.java Renewable.java Book.java DVD.java LibraryDemo.java, then java LibraryDemo. No new imports are required beyond what Chapter 6 already used.

```
$ javac LibraryItem.java Renewable.java Book.java DVD.java LibraryDemo.java
$ java LibraryDemo
```

7.9 OOP in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author -- describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's backend talks to several different external AI answer engines, each with its own API shape, timeout behavior, and response format. Rather than writing separate, duplicated calling code for each one, the backend defines a common interface (this chapter's exact pattern -- a shared contract, no shared implementation) that every provider-specific connector implements, so the rest of the system can call any of them polymorphically without caring which one it happens to be talking to at that moment. Adding a new AI provider later means writing one new class that implements the existing interface, not touching any of the code that already calls it.

7.10 Chapter Summary and Key Takeaways

- An abstract class cannot be instantiated directly; it may mix abstract methods (every subclass must override) with concrete methods (inherited unchanged), and a class may extend only one, abstract or not.
- An interface declares a capability with no state of its own; a class may implement as many interfaces as it needs, and since Java 8 an interface may include default methods with a full, inheritable implementation.
- Rule of thumb: an abstract class for subtypes that share real state and code; an interface for a capability that cuts across any single hierarchy.
- Python's ABC + @abstractmethod mirrors Java's abstract classes closely; typing.Protocol is structural -- a class satisfies it by shape alone, with no inheritance declaration required.
- LibraryItem (abstract) and Renewable (interface) are new this chapter; Book was refactored to use both, and DVD is a brand-new concrete type that shares almost nothing with Book except these two contracts.
- A single loop over List<LibraryItem> calls calculateFine(), renew(), and describe() polymorphically across Book and DVD -- the loop body never checks which concrete type it is holding.

A class hierarchy that only ever holds one kind of object was never really tested -- the moment a second, genuinely different type shows up, an abstract class and an interface are what let old code keep working without a single line of it changing.

7.11 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Why can Java not instantiate LibraryItem directly with new LibraryItem("x", "y"), even though it has a constructor?

2. Add a third LibraryItem subtype, Magazine, that is NOT Renewable at all (magazines in this library can only be returned, never renewed). Which of LibraryItem's methods does Magazine still need to implement, and which interface does it correctly omit?
3. Rewrite Renewable's renewalReceipt default method so that it calls renew() itself instead of taking the outcome as a parameter. Explain the one concrete bug this introduces if a caller ever calls renewalReceipt() more than once.
4. In your own words, explain why Book implementing Renewable and extending LibraryItem at the same time is not a contradiction, even though Java only allows single inheritance.
5. Suppose Library's List<Book> were changed today to List<LibraryItem> without waiting for Chapter 9. Name one existing Library method whose code would need to change, and explain why.

Appendix 7.A — Execution Verification Log

The complete LibraryItem.java, Renewable.java, Book.java, DVD.java, and LibraryDemo.java files (Code/Java/Chapter07/, provided alongside this book) were compiled and executed on OpenJDK 21.0.10 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ javac LibraryItem.java Renewable.java Book.java DVD.java LibraryDemo.java
(no errors)

$ java LibraryDemo
--- Overdue fines after 10 days (calculateFine) ---
"Clean Code" [ISBN 9780132350884] - ON LOAN -> fine: $6.50
"The Imitation Game" [ISBN 9900000000001] - ON LOAN -> fine: $7.50

--- Renewal attempts (Renewable) ---
Clean Code: Renewal attempt: SUCCESS -- Renewed once
The Imitation Game: Renewal attempt: SUCCESS -- No renewals left

--- Full item details (toString, type-specific) ---
"Clean Code" by Robert C. Martin [ISBN 9780132350884, 2008] - ON LOAN (renewed
1x)
"The Imitation Game" [ISBN 9900000000001, 114 min] - ON LOAN (renewed 1x)
```

References

[1] Oracle Corporation. "Abstract Methods and Classes." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html (accessed August 2026).

[2] Oracle Corporation. "Defining Methods in an Interface" and "Default Methods." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html (accessed August 2026).

[3] Python Software Foundation. "abc — Abstract Base Classes." Python 3 documentation. https://docs.python.org/3/library/abc.html (accessed August 2026).

[4] J. Bloch. Effective Java, 3rd ed. Addison-Wesley, 2017, Chapter 20 ("Prefer interfaces to abstract classes").

CHAPTER 8 · EXERCISE CHAPTER

Practice Problems: The Midterm — Reviewing Lectures 1 Through 7

No new material here — eight problems spanning everything from encapsulation to interfaces, designed to make sure the first half of the course actually stuck, before the Midterm.

Learning Objectives — after working through this chapter you will be able to:

(1) Design a small encapsulated class that stores one value and derives others from it on demand. (2) Write control-flow logic that behaves correctly at every boundary value, not just the obvious cases. (3) Build a formatted string efficiently and select a subset of a list while preserving its original order. (4) Define and raise a custom exception, and read and write a plain-text file safely. (5) Design an abstract class where a concrete method is built on top of an abstract one, and an interface with a default method, then dispatch over a list of the interface type polymorphically.

8.1 Recap: Lectures 1–7 in Brief

The first seven lectures built up the whole of the language's day-to-day toolkit. Lecture 1 introduced the shift from procedural to object-oriented thinking. Lecture 2 gave that idea a precise anatomy — fields, constructors, and encapsulation. Lecture 3 filled in Java's primitive types, control flow, Strings, and basic debugging tools. Lecture 4 was Quiz 1, a first checkpoint on all of that. Lecture 5 introduced arrays, the List collection, and the StringBuilder/StringBuffer story for building text efficiently. Lecture 6 added exception handling and file I/O — the tools a program needs once it must survive bad input and outlive a single run. Lecture 7 introduced abstract classes and interfaces, letting Pustaka's Book and a brand-new DVD type share a contract without sharing an implementation (Figure 8.1).

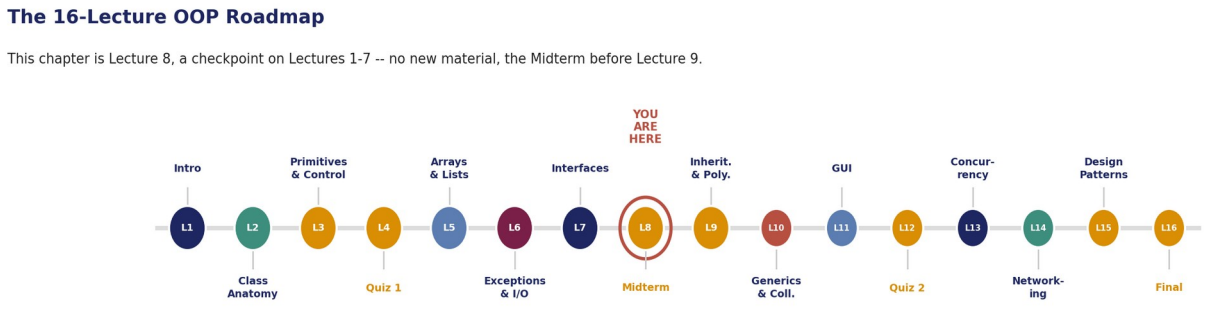


Figure 8.1 — This chapter sits at Lecture 8 in the sixteen-lecture OOP roadmap: a checkpoint, not a new topic, the Midterm before Lecture 9.

8.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a Bab Latihan: a practice chapter, not a new-content chapter. Every problem below is anchored to specific material from Lectures 1–7 (see Figure 8.2). Each problem comes with a HINT — a nudge toward the right way of thinking — but deliberately NOT a full worked solution or source code. These eight problems are standalone, independent of the Pustaka case study that runs through the ordinary chapters — exactly like Quiz 1's problems in Chapter 4.

Where Each Practice Problem Comes From

Every problem in Section 8.3 traces back to one of the previous seven lectures.

#	Problem	Traces back to
1	Temperature Class	L2 -- fields, constructor, a derived value
2	Grade Converter	L3 -- control flow, boundary conditions
3	Roster Formatter	L5 -- StringBuilder, arrays
4	Top-N Scores	L5 -- Lists, order-preserving selection
5	Score Range Check	L6 -- a custom checked exception
6	Score Log File	L6 -- file I/O, try-with-resources
7	Employee Pay	L7 -- abstract class, a concrete method built on an abstract one
8	Discountable Items	L7 -- interface, default method, polymorphic dispatch

Figure 8.2 — Every problem in Section 8.3 traces back to one of the previous seven lectures.

Before you start

Try each problem for real in a fresh .java project before reading its hint. If you get stuck, read only the hint — not a solution — and try again. This is exactly the discipline Section 8.4's Midterm will reward: the exam is open-book, but that is not a substitute for your own reasoning.

8.3 Practice Problems

8.3.1 Fields, Constructors & Encapsulation Review

Problem 1 *[Easy]*. Define a class Q81Temperature(double celsius), storing only a Celsius value, with getCelsius(), getFahrenheit() (F = C * 9/5 + 32), and getKelvin() (K = C + 273.15).

Hint

Store only the celsius field. Both getFahrenheit() and getKelvin() should compute their result on demand from that one field, rather than storing two more fields that could fall out of sync if celsius were ever changed — exactly the same discipline as Chapter 4's Weight class.

8.3.2 Primitives & Control Flow Review

Problem 2 *[Easy]*. Write a class Q82GradeConverter with a static method String letterGrade(double score) returning "A" for score >= 85, "B" for score >= 70, "C" for score >= 55, "D" for score >= 40, and "E" otherwise. Test at 85.0, 84.99, 70.0, 69.99, 55.0, 54.99, 40.0, and 39.99.

> **Hint**
>
> Order your comparisons from highest cutoff to lowest and return as soon as one matches — an if/else-if chain, not a chain of independent ifs. The 84.99/69.99/54.99/39.99 cases exist specifically to catch an off-by-one mistake with >= versus > that the round numbers alone would never reveal.

8.3.3 Arrays, Lists & StringBuilder

Problem 3 *[Medium]*. Write a static method String formatRoster(String[] names) that joins the names with ", " except that the last two are joined with " and " instead — e.g. {"Ann", "Budi", "Citra"} becomes "Ann, Budi, and Citra". Handle 0, 1, 2, and 3-or-more names correctly, and build the result with a StringBuilder rather than repeated String concatenation.

> **Hint**
>
> Handle the 0/1/2-name cases as their own short-circuit branches first — they don't need a loop at all. For 3 or more, append every name except the last two followed by ", ", then append the second-to-last name, the literal ", and ", and finally the last name.

Problem 4 *[Medium]*. Write a static method List<Integer> topN(List<Integer> scores, int n) that returns the n largest values in scores, but in their ORIGINAL relative order from the input list, not sorted by value. Example: scores = [70, 95, 60, 88, 99], n = 3 -> [95, 88, 99] (the three largest values, 95/88/99, kept in the order they originally appeared).

> **Hint**
>
> Don't sort the list you return — sort a COPY to find the value at the cutoff position (the nth-largest value), then walk the ORIGINAL list in its original order and keep every element that clears that cutoff, being careful with duplicate values so you don't keep more than n elements when several share the cutoff value.

8.3.4 Exception Handling & File I/O

Problem 5 *[Medium]*. Define a custom checked exception class InvalidScoreException, and a static method int validateScore(int score) throws InvalidScoreException that returns score unchanged if it is between 0 and 100 inclusive, and otherwise throws an InvalidScoreException whose message includes the invalid value.

> **Hint**
>
> InvalidScoreException should extend Exception (a checked exception, exactly like Chapter 6's BookNotFoundException), giving it a constructor that accepts a message and passes it to super(message). validateScore() only needs one if condition covering both out-of-range directions at once.

Problem 6 *[Medium]*. Write two static methods: void appendScoreRecord(String path, String name, int score) throws IOException, which appends one line "name,score" to the file at path, and List<String[]> readScoreRecords(String path) throws IOException, which reads every line back and returns each as a two-element String array split on the comma.

> **Hint**
>
> Use try-with-resources with a BufferedWriter opened in append mode for the first method, and a BufferedReader for the second — exactly the pattern Chapter 6's saveToFile/loadFromFile used for Pustaka. Don't forget that append mode must not overwrite the file's existing lines.

8.3.5 Interfaces & Abstract Classes

Problem 7 *[Hard]*. Define an abstract class Q87Employee with a name field, an abstract method double monthlyPay(), and a CONCRETE method double annualPay() that returns monthlyPay() * 12. Then define two concrete subclasses: Q87SalariedEmployee (a fixed monthly salary) and Q87CommissionEmployee (a base monthly amount plus a commission rate multiplied by a monthly sales figure).

> **Hint**
>
> annualPay() belongs on the abstract class itself, not on either subclass — it is already fully implementable in terms of whatever monthlyPay() each subclass eventually provides, exactly like Chapter 7's LibraryItem.describe() being a concrete method built on top of the abstract calculateFine().

Problem 8 *[Hard]*. Define an interface Discountable with a method double discountedPrice() and a Java 8+ default method String describeDiscount() built entirely from discountedPrice(). Then define Q88ClearanceItem(double originalPrice) (discountedPrice() = originalPrice * 0.5) and Q88MemberItem(double originalPrice, double memberDiscountRate) (discountedPrice() = originalPrice * (1 - memberDiscountRate)), both implementing Discountable. Finally, sum discountedPrice() over a List<Discountable> containing a mix of both types.

> **Hint**
>
> describeDiscount() should call this.discountedPrice() itself and format the result — it needs no fields of its own, exactly like Chapter 7's Renewable.renewalReceipt(). The summing loop should be written entirely in terms of the Discountable interface type, the same polymorphic-dispatch shape as Chapter 7's List<LibraryItem> loop over Book and DVD.

8.4 Midterm Format: Open-Book, Individual, Take-Home

The Midterm is an open-book, individual, take-home assessment covering exactly the eight kinds of problems reviewed in this chapter — nothing beyond it. Grading is per-problem, not all-or-nothing: partial credit is given for a class that compiles and handles the general case correctly even if one edge case is missed.

Open-book means references, not collaborators

You may consult the textbook, your own notes, and lecture slides while answering. You may NOT collaborate with classmates or submit code that is not genuinely your own, and every submission must include a signed Academic Integrity Pledge — an open-book take-home exam tests whether you can APPLY what you have studied, unsupervised.

Criterion	What it looks for	Weight
Correctness of output	Does the class or method produce the exact expected result for every given test case, including boundary values?	45%
Class & interface design	Are fields private, is state reachable only through methods, and is the abstract/interface split (Problems 7-8) used correctly?	25%
Exception & I/O safety	Does Problem 5's exception carry a useful message, and does Problem 6's file I/O use try-with-resources correctly?	15%
Code clarity & compiles cleanly	Is the code easy to read, sensibly named, and free of dead code — and does it compile with javac with zero errors?	15%

8.5 OOP in Practice: Profitina

About this box

As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

This chapter's eight problems, taken together, are close to a miniature version of what a single feature at Profitina actually requires: a small encapsulated class or two (Problem 1), careful boundary-tested logic (Problem 2), efficient text and list handling for a report or API response (Problems 3-4), safe handling of bad input and persisted state (Problems 5-6), and, when several implementations of the same capability need to be swapped in and out — as Profitina's AI-answer-engine adapters do — an interface or abstract class that lets the rest of the system stay oblivious to which concrete type it is holding (Problems 7-8). A Midterm that only tested one of these skills would miss how often real features need several of them working together in the same small piece of code.

8.6 Chapter Summary

- This chapter introduced no new material — it is a checkpoint covering Lectures 1 through 7.
- Eight standalone problems span the full range reviewed: encapsulation (L2), control-flow boundaries (L3), StringBuilder and Lists (L5), custom exceptions and file I/O (L6), and abstract classes and interfaces (L7).
- Each problem includes a hint, not a solution — working through the reasoning and the code yourself is the point.
- The Midterm is an open-book, individual, take-home assessment: references are allowed, but the code must be your own, correctness on every given test case (including boundary values) is the largest share of the grade, and a signed Academic Integrity Pledge is required with every submission.

A class that compiles is not the same as a class that is correct — and a class that is correct for the round numbers is not the same as one that is correct for the value sitting right at the boundary.

Appendix 8.A — Self-Check Checklist (Non-Graded)

Before submitting any answer — in this chapter's problems or in the Midterm itself — run it through this checklist. It costs a few minutes and catches most of the mistakes graders see most often.

- Does every class and method compile with javac with zero errors and zero warnings?
- Is every field private, with every value the caller needs exposed only through a method?
- Did I test Problem 2's exact boundary values (84.99, 69.99, ...), not just the round numbers that a >= versus > bug could hide behind?
- Does Problem 4's topN() return values in their ORIGINAL order, not sorted, and never more than n elements when values tie at the cutoff?
- Does Problem 5's exception message actually include the invalid value, and does Problem 6's file I/O open its streams with try-with-resources rather than a manual close()?
- Are Problem 7's abstract method and concrete method on the SAME abstract class, with the concrete method calling the abstract one rather than duplicating logic in each subclass?
- Did I reread my own code as if I were a skeptical grader looking for the one input that breaks it?

References

[1] This exercise chapter's assessment format (open-book, individual, take-home, with a signed Academic Integrity Pledge) is modeled on this course's real Midterm Exam (EF234302 Object-Oriented Programming). Its eight practice problems were designed fresh for this rebuild's actual Lecture 1-7 scope — the original exam's problems center on recursive tree and linked-list algorithms, material this rebuild deliberately places in the Data Structures course rather than OOP (see Chapter 1's scope notes), so reusing them here would have tested material this course never taught.

[2] Oracle Corporation. "Abstract Methods and Classes" and "Defining Methods in an Interface." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/ (accessed August 2026).

CHAPTER 9

Inheritance & Polymorphism

Chapter 7 introduced LibraryItem, Renewable, and a brand-new DVD type, but deliberately left two questions unanswered: how deep can an inheritance chain actually go, and when does Library itself stop being Book-specific? This chapter answers both -- adding a third rung to the hierarchy, formalizing what polymorphism actually means at the level of the runtime, and finally generalizing Library to hold any LibraryItem at all.

Learning Objectives — after this chapter you will be able to:

(1) Explain the difference between a class's declared (static) type and its runtime (dynamic) type, and why method calls dispatch on the latter. (2) Write a subclass that extends an already-concrete class (not just an abstract one), and decide correctly whether an override should call super or replace the inherited behavior entirely. (3) Override equals() and hashCode() together, correctly, and explain why overriding one without the other is a bug even though Java's compiler allows it. (4) Explain why reconstructing a polymorphic collection from a flat file needs an explicit type tag, even though the collection itself works polymorphically once built. (5) Generalize a type-specific collection class to hold a common supertype instead, updating every affected method without changing its public behavior for existing callers.

9.1 Recap: Pustaka So Far

Chapter 7 gave Pustaka its first real hierarchy: LibraryItem, an abstract superclass holding title, isbn, and onLoan plus an abstract calculateFine() every concrete subtype must supply; Renewable, an interface for the loan-extension capability; and DVD, a brand-new concrete type sharing nothing with Book except those two contracts. Chapter 8 was a checkpoint, not new material. Library, meanwhile, has quietly stayed exactly as Chapter 5 left it -- a List<Book>, unable to hold the DVD Chapter 7 just introduced. This chapter is where that finally changes.

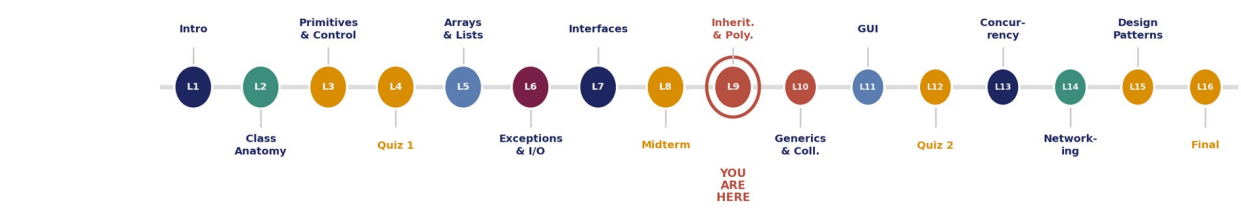


Figure 9.1 — The 16-lecture OOP roadmap. This chapter is Lecture 9: inheritance chains three levels deep, and Library finally holds any LibraryItem.

9.2 Inheritance, Formally: extends, super, and the Override Contract

A class that extends another declares an is-a relationship: every Book is-a LibraryItem, and (as of this chapter) every ReferenceBook is-a Book, which makes it transitively a LibraryItem too. A subclass

constructor's first statement is either an explicit call to super(...), forwarding arguments up the chain, or an implicit call to the superclass's no-argument constructor if none is written -- there is no third option. Inside an overriding method, super.methodName(...) invokes the specific version the superclass defined, which is exactly how Book.borrow() (Chapter 7) reuses LibraryItem's own "already on loan?" check before adding its book-specific step (Figure 9.1).

Book.borrow() -- an override that calls super (Chapter 7, unchanged)

```
@Override
public boolean borrow() {
    if (!super.borrow()) { return false; }
    timesRenewed = 0;
    return true;
}
```

Overriding does not require calling super

Nothing about @Override or the override mechanism itself forces a method to call its parent's version -- calling super is a choice the override makes when it wants to extend, rather than replace, the inherited behavior. Section 9.3 shows the opposite choice in the very same hierarchy.

9.3 A Third Rung: ReferenceBook and Multi-Level Inheritance

ReferenceBook extends Book directly, not LibraryItem -- making LibraryItem → Book → ReferenceBook a genuine three-level chain, the first time this course's case study has gone past one level. ReferenceBook inherits every field and method Book has, including ones it will never use (timesRenewed, renewalStatusLabel()'s tiered wording), and overrides exactly four methods to enforce one rule: a reference item never leaves the building (Figure 9.2).

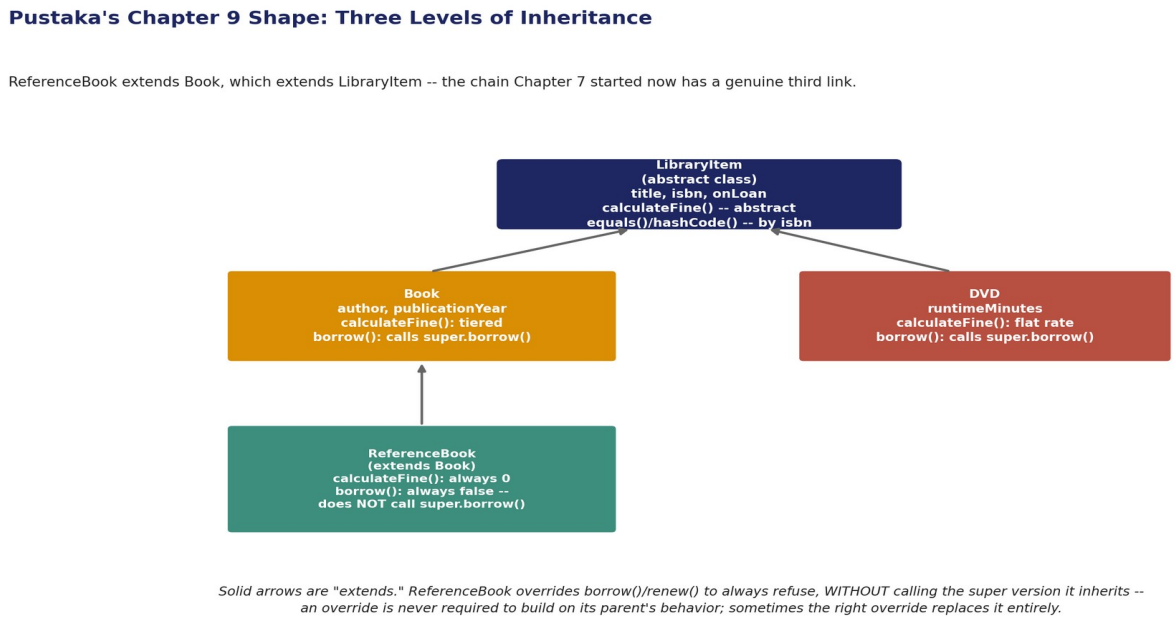


Figure 9.2 — Pustaka's Chapter 9 shape: LibraryItem → Book → ReferenceBook, a genuine third link in the chain Chapter 7 started.

ReferenceBook.java (abridged -- all four overrides shown)

```
public class ReferenceBook extends Book {
    public ReferenceBook(String title, String author, String isbn, int
publicationYear) {
        super(title, author, isbn, publicationYear);
    }

    // Deliberately does NOT call super.borrow() -- replaces it entirely.
    @Override
    public boolean borrow() { return false; }

    @Override
    public boolean renew() { return false; }

    @Override
    public String renewalStatusLabel() { return "Reference only -- cannot be
renewed"; }

    @Override
    public double calculateFine(int daysOverdue) { return 0.0; }
}
```

protected fields reach across every level, not just one

title and isbn are declared protected on LibraryItem, two levels above ReferenceBook -- protected access is inherited transitively, so ReferenceBook.toString() can read title and isbn directly, exactly as Book does, without needing LibraryItem to grant that access a second time.

9.4 Polymorphism, Formally: Static Type vs. Runtime Type

A variable declared as LibraryItem item has a static type of LibraryItem, fixed forever at compile time -- but item might hold a Book, a DVD, or a ReferenceBook at runtime, and that is item's dynamic (or runtime) type. When item.calculateFine(10) is called, Java does not look at the static type to decide which calculateFine() to run; it looks at the ACTUAL object's runtime type and dispatches there, every time. This is dynamic dispatch, and it is the entire mechanism behind every polymorphic loop this course has written since Chapter 7.

Assigning a Book or DVD to a LibraryItem-typed variable (upcasting) is always safe and happens automatically -- a Book genuinely is a LibraryItem, no information is lost. Going the other direction (downcasting, treating a LibraryItem you suspect is really a DVD as a DVD) is not automatic and not always safe: Java requires either an explicit cast, which throws ClassCastException at runtime if you guessed wrong, or the instanceof pattern-match form used throughout this book, which checks first and only proceeds if the guess was correct.

Upcasting is automatic; downcasting needs a check

```
LibraryItem item = new DVD("The Imitation Game", "9900000000001", 114); //
upcast, automatic

if (item instanceof DVD d) {         // downcast, checked first
    System.out.println(d.getRuntimeMinutes() + " minutes");
}
```

9.5 Every Class Is Secretly a Subclass of Object: equals() and hashCode()

Every class in Java -- whether it says so or not -- ultimately extends java.lang.Object, and inherits Object's default equals() and hashCode(): identity-based comparisons, equivalent to == and to the object's memory address, respectively. Two separately-constructed Book objects built from the exact same title, author, and ISBN are NOT .equals() under that default, which is rarely what a real program actually wants to know. LibraryItem overrides both this chapter to compare by ISBN instead -- turning "are these the same object?" into "are these the same real-world item?"

Identity Equality vs. Value Equality, Java and Python Side by Side

Every class in both languages inherits an identity-based default -- LibraryItem overrides it in both tracks to compare by ISBN instead.

	Inherited default	LibraryItem's override
Java	Object.equals(): true only if same object (like ==)	equals()/hashCode() by isbn; built on Book(...).equals(Book(...))
Python	object.__eq__: true only if same object (like is)	__eq__/__hash__ by isbn; defining __eq__ alone makes a class UNHASHABLE -- __hash__ must be defined too, or it's silently set to None

Both languages enforce the same rule for a different reason: an object's hash MUST be built from the exact fields equals()/__eq__ compares, or equal objects could land in different hash buckets and silently vanish from a set or dict lookup that should have found them.

Figure 9.3 — Identity equality vs. value equality, Java and Python side by side.

LibraryItem.equals() and hashCode() (abridged)

```
@Override
public boolean equals(Object other) {
    if (this == other) { return true; }
    if (!(other instanceof LibraryItem that)) { return false; }
    return this.isbn.equals(that.isbn);
}

@Override
public int hashCode() {
    return isbn.hashCode();
}
```

Java will not stop you from overriding only one of the two

The contract is absolute -- if a.equals(b) is true, a.hashCode() must equal b.hashCode() -- but Java's compiler enforces none of it. Overriding equals() alone compiles cleanly and runs, right up until two "equal" items land in different buckets of a HashSet or HashMap and a lookup that should succeed silently returns nothing. (Python's runtime is stricter here: defining __eq__ alone makes a class unhashable by default -- see Figure 9.3.)

9.6 Generalizing Library: One Collection, Any LibraryItem

Library's private field changes from List<Book> to List<LibraryItem> this chapter, and every method built on top of it -- addItem (was addBook), removeItem, removeItemOrThrow (was

removeBookOrThrow, and now throws the generalized ItemNotFoundException below), findByTitle, generateReport -- now works across Book, DVD, and ReferenceBook uniformly, with zero type-specific branching in any of them. The one place type-specific code remains genuinely necessary is file persistence: a plain-text line carries no type information of its own, so saveToFile/loadFromFile need an explicit type tag as the first field on each line.

Library.java -- generalized addItem and generateReport (abridged)

```
public class Library {
    private List<LibraryItem> items = new ArrayList<>();

    public void addItem(LibraryItem item) { items.add(item); }

    // Polymorphic on purpose: never checks which concrete type it holds.
    public String generateReport() {
        StringBuilder sb = new StringBuilder();
        for (LibraryItem item : items) {
            sb.append("- ").append(item.describe()).append("\n");
        }
        return sb.toString();
    }
}
```

Library.java -- the type-tagged file format (abridged)

```
private String toLine(LibraryItem item) {
    if (item instanceof ReferenceBook rb) {
        return "REFBOOK|" + rb.getTitle() + "|" + rb.getAuthor() + "|" +
rb.getIsbn();
    } else if (item instanceof Book b) {
        return "BOOK|" + b.getTitle() + "|" + b.getAuthor() + "|" + b.getIsbn();
    } else if (item instanceof DVD d) {
        return "DVD|" + d.getTitle() + "|" + d.getIsbn() + "|" +
d.getRuntimeMinutes();
    }
    throw new IllegalStateException("Unknown LibraryItem subtype: " +
item.getClass());
}
```

Polymorphism helps in memory; reconstruction needs a tag

Once a Book, a DVD, and a ReferenceBook exist as objects, calling describe() on all three polymorphically needs no type information at all. Rebuilding those same three objects from a flat text file is a different problem entirely -- the file format itself must carry a tag saying which constructor to call, because plain text has no notion of "class" on its own.

Chapter 6's BookNotFoundException is renamed ItemNotFoundException this chapter, for the same reason Library's own methods were renamed -- an exception's name should describe what it actually reports on, and "BookNotFoundException" stopped being accurate the moment Library stopped being Book-only.

9.7 Polymorphism in Action: The Driver Program

The driver program builds a Library holding one Book, one DVD, and one ReferenceBook, then exercises every idea this chapter introduces in sequence: polymorphic borrow() (where

ReferenceBook's override silently refuses), a polymorphic report, value-based equals()/hashCode() (a second Book object sharing an ISBN with the first), instanceof narrowing for the one operation that is genuinely DVD-only, and a full save/load round trip through the type-tagged file format.

LibraryDemo.java (abridged)

```
Library library = new Library();
library.addItem(new Book("Clean Code", "Robert C. Martin", "9780132350884",
2008));
library.addItem(new DVD("The Imitation Game", "9900000000001", 114));
library.addItem(new ReferenceBook("Encyclopedia of Computer Science", "Ralston &
Reilly", "9780123456789", 2003));

LibraryItem book = library.findByTitle("Clean Code");
LibraryItem dvd = library.findByTitle("The Imitation Game");
LibraryItem refBook = library.findByTitle("Encyclopedia of Computer Science");

Book copyOfSameBook = new Book("Clean Code (2nd copy)", "Robert C. Martin",
"9780132350884", 2008);
System.out.println(book.equals(copyOfSameBook)); // true -- same ISBN, different
object

for (LibraryItem item : new LibraryItem[] { book, dvd, refBook }) {
    if (item instanceof DVD d) {
        System.out.println(d.getRuntimeMinutes() + " minutes");
    }
}
```

9.8 Compiling and Running Your Program

javac LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java ItemNotFoundException.java Library.java LibraryDemo.java, then java LibraryDemo. No new imports are required beyond what Chapter 6 already used.

```
$ javac LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java \
      ItemNotFoundException.java Library.java LibraryDemo.java
$ java LibraryDemo
```

9.9 OOP in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author -- describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's backend regularly discovers the same real-world brand or entity mentioned across multiple different sources -- different URLs, different exact wording -- and needs to recognize that two records refer to the SAME thing rather than treating them as unrelated. That is exactly this chapter's equals()/hashCode() problem at a larger scale: identity (are these the literal same object in memory, or the same database row) is the wrong question; value equality against a canonical identifier (a

normalized brand name or domain, playing the same role this chapter's isbn does) is the right one, and getting hashCode() wrong the same silent way this chapter warns against would mean duplicate entries quietly surviving deduplication that was supposed to catch them.

9.10 Chapter Summary and Key Takeaways

- A method call dispatches on an object's runtime (dynamic) type, never its declared (static) type -- this is the entire mechanism behind every polymorphic loop since Chapter 7.
- Upcasting (subtype to supertype) is automatic and always safe; downcasting (supertype to subtype) needs an explicit cast or an instanceof pattern match, since the guess can be wrong.
- Overriding a method is never required to call super -- ReferenceBook.borrow() deliberately replaces LibraryItem's inherited behavior instead of extending it.
- Inheritance can chain more than one level deep (LibraryItem → Book → ReferenceBook); a protected field declared at the top is still reachable at the bottom.
- Every class implicitly extends Object and inherits its identity-based equals()/hashCode(); overriding them together (never just one) switches a class over to value-based equality, built from the same field(s) in both methods.
- Library now holds List<LibraryItem>, not List<Book> -- every method is polymorphic except file persistence, which needs an explicit type tag to reconstruct concrete objects from plain text.

A hierarchy is only really tested the day someone asks it to do the one thing it was never built for -- and the day Library finally had to hold a DVD and a ReferenceBook alongside a Book, without a single one of its callers noticing the difference, was that day.

9.11 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. ReferenceBook.borrow() returns false unconditionally and never calls super.borrow(). Rewrite it so it DOES call super.borrow() first, and explain in one sentence why the resulting behavior would be wrong for a reference-only item.
2. Suppose LibraryItem.equals() compared title instead of isbn. Give one concrete pair of Book objects that would become incorrectly "equal" under that change, and explain why isbn is the safer choice.
3. Add a fourth level to the hierarchy: RareBook extends ReferenceBook, adding an estimatedValue field. Which of ReferenceBook's methods, if any, does RareBook need to override to change behavior, versus simply inherit unchanged?
4. Library.toLine() checks `instanceof ReferenceBook` before `instanceof Book`. Explain what would go wrong if that order were reversed, given that every ReferenceBook is also a Book.
5. In your own words, explain the difference between a ClassCastException thrown by an unchecked (LibraryItem) item cast, and the instanceof pattern-match form used throughout this chapter -- which one can actually fail at runtime, and which one cannot?

Appendix 9.A — Execution Verification Log

The complete LibraryItem.java, Book.java, DVD.java, ReferenceBook.java, Renewable.java, ItemNotFoundException.java, Library.java, and LibraryDemo.java files (Code/Java/Chapter09/, provided alongside this book) were compiled and executed on OpenJDK 21.0.10 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ javac LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java \
      ItemNotFoundException.java Library.java LibraryDemo.java
(no errors)

$ java LibraryDemo
--- Borrow attempts (polymorphic dispatch) ---
Clean Code .borrow() -> true
The Imitation Game .borrow() -> true
Encyclopedia .borrow() -> false  (ReferenceBook overrides borrow() to always
refuse -- never calls super.borrow())

--- equals()/hashCode(): value equality, not identity ---
book == copyOfSameBook       -> false  (different objects in memory)
book.equals(copyOfSameBook)  -> true  (same ISBN -> same real-world item)
book.hashCode() == copy's     -> true

--- instanceof pattern matching: the one DVD-only operation ---
Clean Code is not a DVD -- no runtime to report
The Imitation Game runtime: 114 minutes
Encyclopedia of Computer Science is not a DVD -- no runtime to report

--- removeItemOrThrow on a missing ISBN ---
Caught: No library item found with ISBN: 0000000000000
```

References

[1] Oracle Corporation. "Overriding and Hiding Methods." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/override.html (accessed August 2026).

[2] Oracle Corporation. "Polymorphism." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html (accessed August 2026).

[3] Oracle Corporation. "Object." Java SE 21 API Documentation. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Object.html (accessed August 2026).

[4] J. Bloch. Effective Java, 3rd ed. Addison-Wesley, 2017, Item 10 ("Obey the general contract when overriding equals") and Item 11 ("Always override hashCode when you override equals").

CHAPTER 10

Generics & the Collections Framework

Chapter 9 finally let Library hold any LibraryItem, but left it with exactly one access pattern (a linear scan by title) and no guaranteed order across a mix of Books, DVDs, and ReferenceBooks. This chapter adds the Collections Framework's other core structure -- a Map index alongside the List -- plus a reusable generic Pair class, a bounded-wildcard utility method, a Comparable-driven natural ordering, and a Set of distinct concrete types.

Learning Objectives — after this chapter you will be able to:

(1) Explain what a generic class's type parameters (Pair<A, B>) represent, and why type erasure means Pair<String, Integer> and Pair<LibraryItem, Double> share exactly one .class file at runtime. (2) Read and write a bounded wildcard parameter type (List<? extends LibraryItem>), and explain why List<Book> can be passed where it is expected even though List<Book> is not itself a List<LibraryItem>. (3) Add a second Map-based index to an existing List-backed collection class without breaking any existing method, keeping both structures updated together on every insert and removal. (4) Implement Comparable so Collections.sort() produces a natural ordering with no separate Comparator, and explain what that ordering is being derived from. (5) Explain the difference between a Set that preserves insertion order (LinkedHashSet) and one that makes no ordering guarantee (HashSet), and say which one a reproducible log needs.

10.1 Recap: Pustaka So Far

Chapter 9 generalized Library's field from List<Book> to List<LibraryItem> and formalized static vs. runtime type, but every one of Library's methods still does exactly one thing with that List: search it linearly by title. There is still no way to look an item up by ISBN in anything faster than O(n), no guaranteed order when a report mixes a Book, a DVD, and a ReferenceBook, and no way to tell, at a glance, which concrete subtypes a given Library actually contains. This chapter's shape answers all three, using the two other pillars of the Java Collections Framework this course has not yet needed: Map and Set.

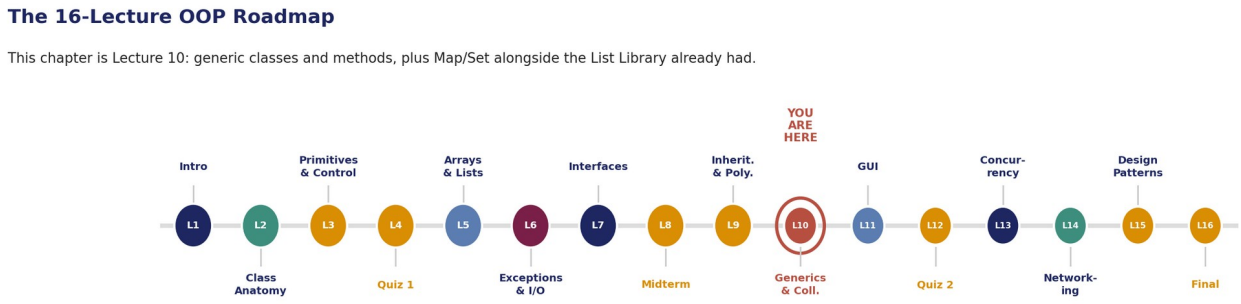


Figure 10.1 — The 16-lecture OOP roadmap. This chapter is Lecture 10: generic classes, a bounded-wildcard method, and a Map/Set-based second index alongside the List Library already had.

10.2 A Reusable Generic Class: Pair<A, B>

Pairing a LibraryItem with a computed value -- its fine amount, say -- or a title with a publication year, is a shape that recurs throughout a program, and writing a one-off class for every combination of types would mean writing the same four lines of boilerplate over and over. Pair<A, B> writes that shape exactly once: A and B are type parameters, not real types -- the compiler substitutes real types in at each call site (Pair<LibraryItem, Double>, Pair<String, Integer>, ...) and checks that every use of that particular Pair is consistent with the types chosen there (Figures 10.1 and 10.2).

Pair.java (complete)

```
public final class Pair<A, B> {
    private final A first;
    private final B second;

    public Pair(A first, B second) {
        this.first = first;
        this.second = second;
    }

    public A getFirst() { return first; }
    public B getSecond() { return second; }

    @Override
    public String toString() {
        return "(" + first + ", " + second + ")";
    }
}
```

Generics, Compared: Compile-Time-Checked vs. Never-Checked

Both languages let one class/function work over many types -- but what actually happens to the type information at runtime is very different.

	Java: List<T>, Pair<A,B>, <T extends X>	Python: list[T], Pair[A, B], T: X
Checked when?	At COMPILE time -- javac rejects a mismatched call before it ever runs	NEVER by the interpreter itself -- only by an optional tool (mypy, pyright) you choose to run separately
What survives to runtime?	Nothing -- type erasure removes all generic info from the .class file; List<String> and List<Integer> share ONE class object at runtime	Nothing enforced either, but for a different reason -- CPython never reads type hints as rules, only as optional metadata
Practical result	A type mismatch is a COMPILE error, caught before the program ever runs	A type mismatch is invisible until it causes a real bug -- or a static checker is run and reports it

Neither language checks generics IN THE RUNNING PROGRAM -- Java checks once, at compile time, then discards the information; Python never checks at all unless a separate tool is asked to. "Java generics are erased" and "Python generics are unenforced" are not the same claim.

Figure 10.2 — Generics, compared: checked once at compile time then erased (Java) vs. never checked by the interpreter at all (Python).

Type erasure: one class, not one per type combination

After javac has checked every use of a Pair<LibraryItem, Double> for consistency, it throws the generic information away -- the compiled Pair.class holds plain Object fields, with no trace of A or B left in it. A Pair<String, Integer> and a Pair<LibraryItem, Double> are, at runtime, literally the same class; pair.getClass() can never tell you what A or B were. The checking still happened -- just once, at compile time, not on every method call the way an if/instanceof check would.

10.3 Bounded Wildcards: List<? extends LibraryItem>

LibraryUtils.totalFines() needs to add up calculateFine() across a List of items, but it should not have to care whether that List holds LibraryItem, just Book, or just DVD -- any of the three is a reasonable argument, since all it ever does is read each item and call a method every LibraryItem guarantees. Writing the parameter as plain List<LibraryItem> would refuse a List<Book> outright: Java's generics are invariant, so a List<Book> is NOT a List<LibraryItem>, even though a Book IS a LibraryItem. The bounded wildcard List<? extends LibraryItem> is exactly the gap this closes -- it accepts a List of LibraryItem or of any subtype, in exchange for only being usable to READ items out (adding to it is deliberately restricted, since the compiler cannot know precisely which subtype it would need to accept).

LibraryUtils.java (complete)

```
public final class LibraryUtils {

    private LibraryUtils() { } // utility class: never instantiated

    public static double totalFines(List<? extends LibraryItem> items, int
daysOverdue) {
        double total = 0.0;
        for (LibraryItem item : items) {
            total += item.calculateFine(daysOverdue);
        }
        return total;
    }
}
```

A bounded wildcard is not the same thing as a generic method

totalFines() is NOT a generic method -- its signature declares no type parameter of its own (no <T> before the return type), only a wildcard inside an ordinary parameter's type. A true generic method here would read public static <T extends LibraryItem> double totalFines(List<T> items, int daysOverdue) instead; either form compiles and works for this use case, but they answer different questions (the wildcard form says "I only need to read LibraryItems, I don't care what T actually is," while a type-parameter form would let the method's body also refer to T by name). Section 10.3 of the Python-track book uses the type-parameter form directly, since Python's newer generic-function syntax makes that the more natural choice there -- see the callout in that section.

10.4 One Collection, Two Indexes: byIsbn Alongside items

Library's items field (a List<LibraryItem>, unchanged since Chapter 9) is joined this chapter by a second field, byIsbn, a Map<String, LibraryItem> keyed on ISBN -- addItem and removeItem now update both collections together, every time, so the two never drift out of sync. findByIsbn(isbn) then answers in O(1) what findByTitle() still answers in O(n): both search the exact same underlying LibraryItem objects, just through two different paths (Figure 10.3).

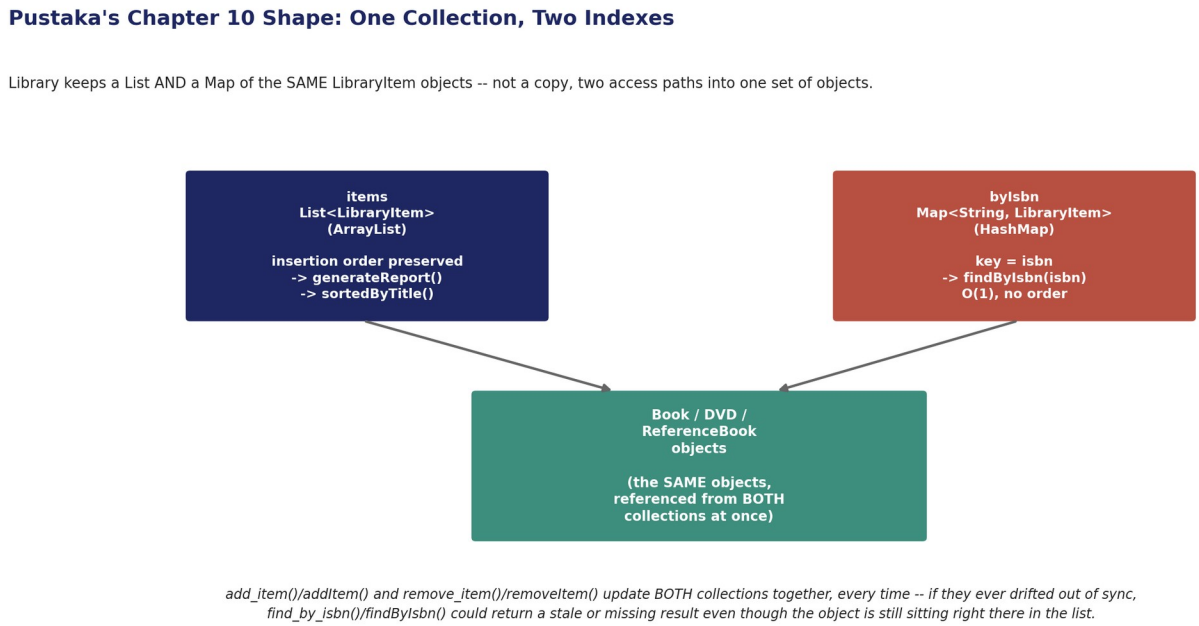


Figure 10.3 — Pustaka's Chapter 10 shape: Library's List index and Map index referencing the same LibraryItem objects.

Library.java -- the byIsbn index (abridged)

```
public class Library {
    private List<LibraryItem> items;
    private Map<String, LibraryItem> byIsbn;

    public Library() {
        this.items = new ArrayList<>();
        this.byIsbn = new HashMap<>();
    }

    public void addItem(LibraryItem item) {
        items.add(item);
        byIsbn.put(item.getIsbn(), item);
    }

    public boolean removeItem(String isbn) {
        for (int i = 0; i < items.size(); i++) {
            if (items.get(i).getIsbn().equals(isbn)) {
                items.remove(i);
                byIsbn.remove(isbn);
                return true;
            }
        }
        return false;
    }

    public LibraryItem findByIsbn(String isbn) {
        return byIsbn.get(isbn);
    }
}
```

Why keep the List at all, once there is a Map?

A HashMap alone would answer findByIsbn() just as well, but it makes no promise whatsoever about iteration order -- generateReport() and this chapter's sortedByTitle() both need a stable, insertion-ordered starting point to work from. Library keeps both structures because they serve two genuinely different jobs: the List is the source of truth for order, the Map is a second, order-blind index built purely for O(1) lookup speed.

10.5 Natural Ordering: LibraryItem implements Comparable<LibraryItem>

Collections.sort(list) (or list.sort(null)) needs to know how to put two LibraryItems in order, and it gets that either from an explicit Comparator argument or from the element type itself implementing Comparable -- Chapter 10 chooses the latter, so every List<LibraryItem> in this program has one obvious default order (by title) without a caller ever having to supply a Comparator just to sort a list.

LibraryItem.java -- compareTo() (new this chapter)

```
public abstract class LibraryItem implements Comparable<LibraryItem> {
    // ... title, isbn, onLoan, equals()/hashCode() unchanged from Chapter 9 ...

    @Override
    public int compareTo(LibraryItem other) {
        return this.title.compareToIgnoreCase(other.title);
    }
}
```

Library.java -- sortedByTitle()

```
public List<LibraryItem> sortedByTitle() {
    List<LibraryItem> copy = new ArrayList<>(items);
    Collections.sort(copy);
    return copy;
}
```

Natural ordering vs. an explicit Comparator

Comparable.compareTo() supplies a class's NATURAL ordering -- the one default order the class itself claims makes sense, here by title, case-insensitively. A caller who wants a different order (by ISBN, by fine amount, by publication year) is never stuck with title: passing an explicit Comparator to Collections.sort(list, comparator) overrides the natural order for that one call, without changing compareTo() or affecting any other caller.

10.6 A LinkedHashSet of Distinct Types

distinctTypes() answers a question Library could not answer in one line before this chapter: which concrete subtypes does this Library actually hold right now? A Set's whole contract is "no duplicates" -- three Books contribute the string "Book" to the result only once, however many Book objects are really present -- and Library deliberately builds that Set as a LinkedHashSet rather than a plain HashSet, specifically so the result comes back in first-seen order every time, run after run.

Library.java -- distinctTypes()

```
public Set<String> distinctTypes() {
    Set<String> types = new LinkedHashSet<>();
    for (LibraryItem item : items) {
        types.add(item.getClass().getSimpleName());
    }
    return types;
}
```

A plain HashSet would not have broken the code -- only the log

Swapping LinkedHashSet for a plain HashSet here would still produce a correct, duplicate-free Set<String>; nothing about the METHOD's correctness depends on which Set implementation is used. What a plain HashSet does not guarantee is which order the three type names print in when the Set is iterated -- and Appendix 10.A's execution log needs that order to be the same on every run for the transcript below to be trustworthy as a REPRODUCTION, not a one-time snapshot. That reproducibility requirement, not correctness, is exactly why LinkedHashSet was chosen here.

10.7 Putting It Together: The Driver Program

The driver builds a Library holding four items -- the Chapter 9 trio plus a fourth Book, Sedgewick's Algorithms -- then exercises this chapter's five additions in sequence: an O(1) findByIsbn() lookup, sortedByTitle() feeding directly into the bounded-wildcard totalFines(), distinctTypes() (Section 10.6's excerpt already shows this one), and finally two Pair<A, B> instances built from the same data. The abridged extract below shows the first, third, and fifth of those directly; the complete sequence, plus the Chapter 9 recap steps that precede it, is in Code/Java/Chapter10/LibraryDemo.java.

LibraryDemo.java (abridged)

```
Library library = new Library();
library.addItem(new Book("Clean Code", "Robert C. Martin", "9780132350884",
2008));
library.addItem(new DVD("The Imitation Game", "9900000000001", 114));
library.addItem(new ReferenceBook("Encyclopedia of Computer Science", "Ralston &
Reilly", "9780123456789", 2003));
library.addItem(new Book("Algorithms", "Robert Sedgewick", "9780321573513",
2011));

LibraryItem foundByIsbn = library.findByIsbn("9780321573513");
System.out.println(foundByIsbn.getTitle());

List<LibraryItem> byTitle = library.sortedByTitle();
double allFines = LibraryUtils.totalFines(byTitle, 10);
System.out.printf("$%.2f%n", allFines);

LibraryItem book = library.findByTitle("Clean Code");
Pair<LibraryItem, Double> mostExpensive = new Pair<>(book,
book.calculateFine(10));
System.out.println(mostExpensive);
Pair<String, Integer> titleAndYear = new Pair<>("Clean Code", 2008);
System.out.println(titleAndYear);
```

10.8 Compiling and Running Your Program

javac LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java ItemNotFoundException.java Pair.java LibraryUtils.java Library.java LibraryDemo.java, then java LibraryDemo. No new imports are required beyond java.util.Map, java.util.HashMap, java.util.LinkedHashSet, and java.util.Set, all part of the same java.util package Chapter 5 already imported ArrayList and List from.

```
$ javac LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java \
      ItemNotFoundException.java Pair.java LibraryUtils.java Library.java
LibraryDemo.java
$ java LibraryDemo
```

10.9 OOP in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author -- describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's backend keeps an ordered list of tracked brand mentions for its ranking reports, and, alongside it, a hash-indexed map keyed by a canonical external identifier for O(1) deduplication lookups when a new mention arrives -- precisely this chapter's items/byIsbn shape, for precisely the same reason: one structure preserves the order a report needs, the other trades order away for lookup speed. A shared, generic pairing utility (this chapter's Pair, in spirit) shows up wherever the backend needs to carry a computed score alongside the record it was computed from, without writing a bespoke two-field class for every score type it ever adds.

10.10 Chapter Summary and Key Takeaways

- Generic type parameters (Pair<A, B>) are checked for consistency once, at compile time, then erased -- at runtime there is exactly one Pair.class, shared by every A/B combination ever used.
- A bounded wildcard (List<? extends LibraryItem>) is not a generic method -- it lets an ordinary method accept a List<LibraryItem> or any List<subtype>, in exchange for only reading from that list, since Java's generics are invariant and would otherwise reject List<Book> outright.
- Library now keeps a Map<String, LibraryItem> (byIsbn) alongside its List<LibraryItem> (items), updated together on every insert and removal, trading the Map's lack of ordering for O(1) ISBN lookup.
- Implementing Comparable gives a class a natural ordering that Collections.sort() uses with no Comparator argument; an explicit Comparator can still override that default for any one call, without touching compareTo().
- A LinkedHashSet, not a plain HashSet, is what makes distinctTypes()'s output order reproducible across runs -- both are equally correct as Sets, but only one guarantees the iteration order a verification log can rely on.

A collection class earns the name "framework" the day it stops being just a List with extra methods bolted on -- and the day Library grew a Map that had to stay in lockstep with its List, on every single insert and removal, without either one ever being allowed to fall behind, was that day.

10.11 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. LibraryUtils.totalFines() takes a List<? extends LibraryItem> rather than a List<LibraryItem>. Give one call site from LibraryDemo.java that would fail to compile if the parameter were changed to plain List<LibraryItem>, and explain why.

2. Suppose Library.removeItem(isbn) forgot to call byIsbn.remove(isbn), removing the item from items only. Describe a concrete sequence of calls after that point that would return a wrong (but not obviously wrong) answer.
3. Rewrite LibraryUtils.totalFines() as a true generic method with a bounded type parameter (public static <T extends LibraryItem> double totalFines(List<T> items, int daysOverdue)) instead of a wildcard. Would every existing call site in LibraryDemo.java still compile unchanged? Explain your answer.
4. LibraryItem.compareTo() orders by title, case-insensitively. Write an alternative Comparator<LibraryItem> that orders by calculateFine(10) descending instead, and give one line of code that would use it without changing compareTo() at all.
5. Explain, in your own words, why Pair.toString() can safely write "(" + first + ", " + second + ")" without ever needing to know, at compile time or at runtime, what concrete types A and B turned out to be.

Appendix 10.A — Execution Verification Log

The complete LibraryItem.java, Renewable.java, Book.java, DVD.java, ReferenceBook.java, ItemNotFoundException.java, Pair.java, LibraryUtils.java, Library.java, and LibraryDemo.java files (Code/Java/Chapter10/, provided alongside this book) were compiled and executed on OpenJDK 21.0.10 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ javac LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java \
      ItemNotFoundException.java Pair.java LibraryUtils.java Library.java
LibraryDemo.java
(no errors)

$ java LibraryDemo
--- Borrow attempts (polymorphic dispatch, Chapter 9) ---
Clean Code .borrow() -> true
The Imitation Game .borrow() -> true
Encyclopedia .borrow() -> false  (ReferenceBook overrides borrow() to always
refuse -- never calls super.borrow())

--- equals()/hashCode(): value equality, not identity (Chapter 9) ---
book.equals(copyOfSameBook)  -> true  (same ISBN -> same real-world item)

--- NEW Chapter 10: findByIsbn(), the Map-backed O(1) lookup ---
findByIsbn("9780321573513") -> Algorithms

--- NEW Chapter 10: sortedByTitle(), via LibraryItem implements Comparable ---
  Algorithms
  Clean Code
  Encyclopedia of Computer Science
  The Imitation Game

--- NEW Chapter 10: distinctTypes(), a Set<String> (no duplicates) ---
distinctTypes() -> [Book, DVD, ReferenceBook]

--- NEW Chapter 10: LibraryUtils.totalFines(), using a bounded wildcard (List<?
extends LibraryItem>) ---
totalFines(all items, 10 days overdue) -> $20.50

--- NEW Chapter 10: Pair<A, B>, a generic class ---
Pair<LibraryItem, Double> -> ("Clean Code" by Robert C. Martin [ISBN
9780132350884, 2008] - ON LOAN (renewed 0x), 6.5)
Pair<String, Integer>     -> (Clean Code, 2008)

--- removeItemOrThrow on a missing ISBN (Chapter 6/9) ---
Caught: No library item found with ISBN: 0000000000000

--- Save/load round trip (type-tagged file format, Chapter 6/9) ---
Reloaded 4 item(s), distinct types: [Book, DVD, ReferenceBook]
```

References

[1] Oracle Corporation. "Generics (Updated)." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/generics/index.html (accessed August 2026).

[2] Oracle Corporation. "Type Erasure." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/generics/erasure.html (accessed August 2026).

[3] Oracle Corporation. "LinkedHashSet<E>." Java SE 21 API Documentation. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedHashSet.html (accessed August 2026).

[4] J. Bloch. Effective Java, 3rd ed. Addison-Wesley, 2017, Item 26 ("Don't use raw types") and Item 31 ("Use bounded wildcards to increase API flexibility").

CHAPTER 11

GUI Programming: Events & MVC

Every prior chapter's Pustaka has been a console program: a driver method that runs top to bottom once, prints its results, and exits. This chapter gives Pustaka a real windowed interface -- a list of items, a form to add a book, buttons to borrow/return/remove -- built with JavaFX and organized with the Model-View-Controller (MVC) pattern, which splits the program into a Model that knows nothing about any GUI, a View that knows nothing about library logic, and a Controller that is the only class allowed to touch both (Figure 11.1).

Learning Objectives — after this chapter you will be able to:

(1) Explain what the Model-View-Controller pattern separates, and state which of the three classes is allowed to depend on which others. (2) Read and write a JavaFX event handler bound with setOnAction(...), and explain when JavaFX actually calls it. (3) Build a JavaFX View class whose only job is constructing and exposing controls, with zero business logic of its own. (4) Trace a single button click all the way through: View exposes the click -> Controller's handler reads the View -> Controller mutates the Model -> Controller tells the View to refresh. (5) Explain why a headless, scripted verification run (selecting a row and firing a button programmatically) proves the same code path a real mouse click would take, and is not a simulation of the GUI.

11.1 Recap: Pustaka So Far

Chapter 10 gave Library a second index (a Map<String, LibraryItem> keyed on ISBN, alongside its List<LibraryItem>), a reusable generic Pair<A, B> class, a bounded-wildcard utility method, and a natural ordering via Comparable -- but every chapter through Chapter 10 still interacts with that Library through nothing more than a sequence of System.out.println() calls chosen once, at compile time, by the programmer. Nothing in LibraryDemo.java lets a person choose which item to borrow while the program is running; the driver decides everything in advance. This chapter's shape answers that: a real window, where a person clicks an actual list row and an actual button, and the program responds to whatever they clicked, not to a script written in advance.

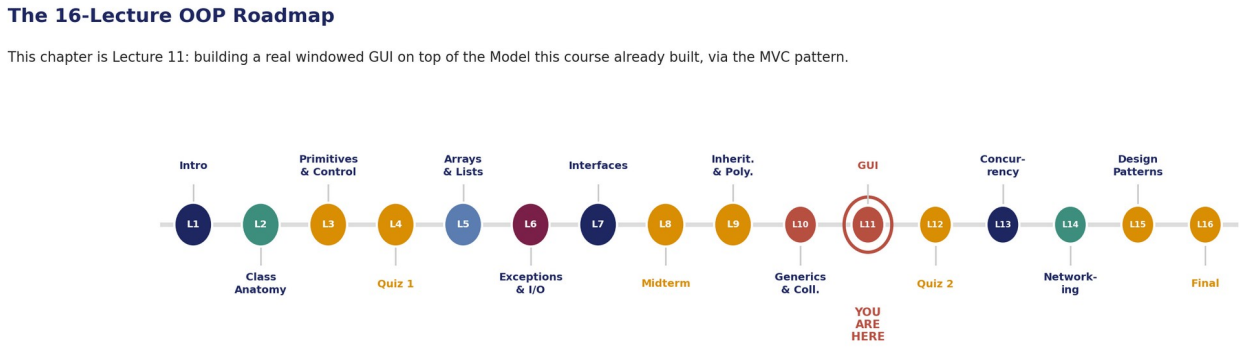


Figure 11.1 — The 16-lecture OOP roadmap. This chapter is Lecture 11: a real windowed GUI built on the same Library/LibraryItem Model this course has built since Chapter 5, via the Model-View-Controller pattern.

11.2 From the Console to a Window: Event-Driven Programming

A console driver like LibraryDemo.java is a program IN CONTROL: it decides, line by line, what happens next, and calling System.in for input (if it did at all) would only ever pause that same fixed sequence, never change its shape. A GUI program inverts that relationship entirely -- once LibraryApp.start() finishes building the window and returns, the program's own code stops driving anything. JavaFX's event loop takes over, sits idle until the user does something (clicks a button, types into a field, closes the window), and calls back into exactly one small piece of the programmer's own code for each such event. This style is called EVENT-DRIVEN PROGRAMMING: the program's logic is a collection of callbacks, each one short, each one triggered by something the user did, with no method anywhere that describes the whole run from start to finish the way main() used to.

> **The event loop never blocks waiting for YOUR code**
>
> JavaFX's event loop calls a handler, waits for it to return, then goes back to waiting for the next event -- it does not multitask your handler against anything else. This is exactly why every handler in this chapter's LibraryController is short and returns immediately: a handler that ran for a long time (a slow network call, say) would freeze the entire window for that same duration, since nothing else -- not even redrawing the window -- can happen while the event loop is inside your callback. (Chapter 13's concurrency material is precisely the tool for handlers that genuinely need to do slow work without freezing the UI; this chapter's handlers are all fast enough that the question never comes up.)

11.3 The Model-View-Controller Pattern

MVC splits a GUI program into three classes with three separate jobs, and exactly one allowed relationship between them: the Controller may depend on both the Model and the View, but the Model must never import or reference anything about the View, and the View must never call anything on the Model directly. Pustaka's Model -- the Library and LibraryItem hierarchy -- was written across Chapters 5 through 10 with no GUI in mind at all, and this chapter needed to add exactly one method to it (getAllItems(), a defensive-copy accessor) to make it usable from a View. Every other Model method Chapters 5-10 already wrote -- addItem(), removeItem(), findByTitle(), borrow(), returnItem() -- is reused completely unchanged (Figure 11.2).

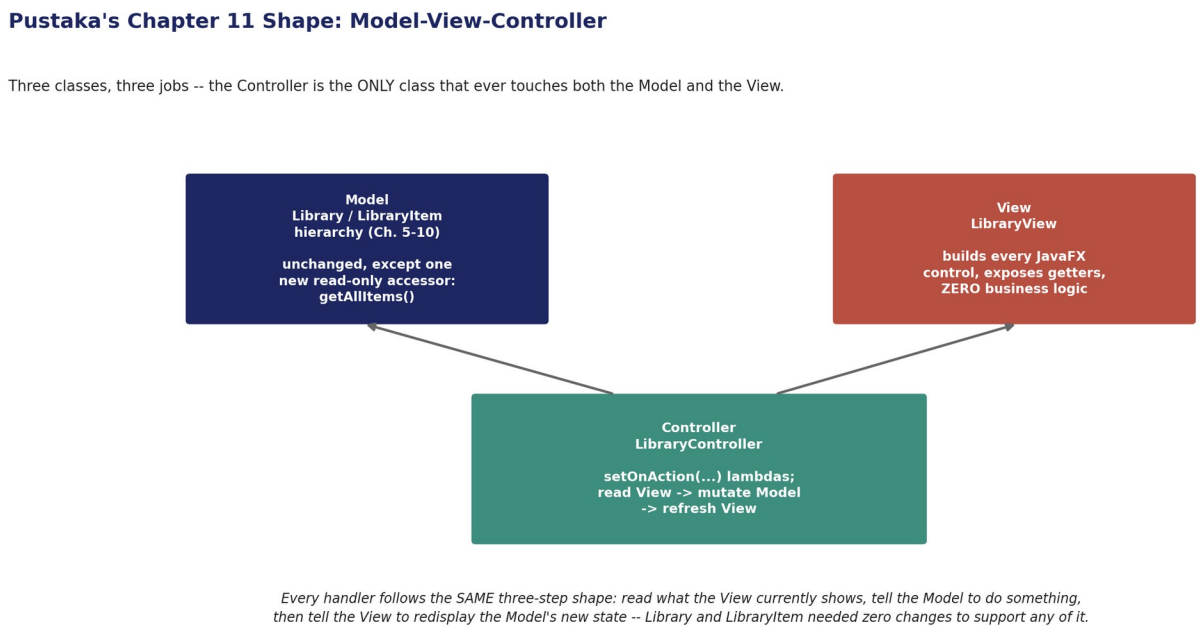


Figure 11.2 — Pustaka's Chapter 11 shape: Model, View, and Controller, with the Controller as the only class touching both of the other two.

> **Model-View-Controller, precisely**
>
> MODEL: the data and the rules about it (Library, LibraryItem, Book, DVD, ReferenceBook) -- knows nothing about buttons, windows, or pixels. VIEW: the visible controls and their layout (LibraryView) -- knows nothing about what borrowing or removing an item actually means, only how to display a LibraryItem and expose its own controls. CONTROLLER: the glue (LibraryController) -- reads what the View currently shows, tells the Model what to do, then tells the View to redisplay the Model's new state. A View built against a completely different Library would look pixel-for-pixel identical; a Model used by a completely different View would behave identically.

11.4 The View: LibraryView

LibraryView builds every visible JavaFX control in its constructor -- a ListView<LibraryItem> for the items, four TextFields for adding a book, four buttons, and a status Label -- and exposes each one through a public getter. It calls nothing on Library, and it contains no method that decides what borrowing, adding, or removing actually means; its only substantive method besides the getters is refresh(List<LibraryItem>), which simply redraws whatever list of items it is handed.

LibraryView.java (excerpt)

```
public class LibraryView {
    private final ListView<LibraryItem> itemList = new ListView<>();
    private final Button borrowButton = new Button("Borrow Selected");
    private final Button returnButton = new Button("Return Selected");
    private final Button removeButton = new Button("Remove Selected");
    private final Label statusLabel = new Label("Ready.");
    // ... titleField/authorField/isbnField/yearField, addButton ...

    public LibraryView() {
        itemList.setCellFactory(list -> new ListCell<>() {
            @Override
            protected void updateItem(LibraryItem item, boolean empty) {
                super.updateItem(item, empty);
                setText(empty || item == null ? null : item.describe());
            }
        });
        // ... build GridPane form, HBox/VBox layout,
root.setCenter()/setRight() ...
    }

    public ListView<LibraryItem> getItemList() { return itemList; }
    public Button getBorrowButton() { return borrowButton; }
    // ... one getter per control ...

    public void refresh(List<LibraryItem> items) {
        LibraryItem selected = itemList.getSelectionModel().getSelectedItem();
        itemList.getItems().setAll(items);
        if (selected != null && items.contains(selected)) {
            itemList.getSelectionModel().select(selected);
        }
    }

    public void setStatus(String message) { statusLabel.setText(message); }
}
```

A cell factory is how a ListView displays custom objects

By default, a JavaFX ListView<LibraryItem> would call toString() on each item to decide what text to show -- which would work, but couples the View to whichever toString() a Model class happens to have. The cellFactory above instead calls describe() explicitly, the exact same method every LibraryItem already exposes for exactly this purpose (Chapters 7-9) -- the View chooses how to display an item, rather than quietly inheriting whatever the Model's toString() happens to produce.

11.5 The Controller: Wiring Events

LibraryController is the only class in this chapter that imports both Library and LibraryView. Its constructor calls wireEvents() (attaching one setOnAction(...) lambda per button) and then refreshView() once, so the window opens already showing the Model's starting state. Every handler follows the same three-step shape: read the View's current selection or form fields, tell the Model to do something, then call refreshView() so the View redisplays whatever the Model's state now is (Figure 11.3).

Event Handling, Compared: Two Toolkits, One Idea

Both GUI toolkits call YOUR code only in response to a real user action -- but how a callback is attached, and what it receives, differs.

	Java: JavaFX	Python: Tkinter
Binding a callback	button.setOnAction(e -> handler()) a lambda implementing EventHandler<ActionEvent>	button.config(command=handler) a plain zero-argument callable
What the callback receives	An ActionEvent object (often unused, but always passed)	Nothing -- Tkinter's command= callback takes no arguments at all
Who runs the event loop	Application.launch() starts JavaFX's own rendering + event thread	root.mainloop() starts Tk's own event loop on the calling thread

Neither toolkit calls your handler code until the user actually clicks -- both hand control BACK to the toolkit the instant your handler returns. "An event handler" means the same idea in both trees; the object it receives (or doesn't) and who owns the loop are toolkit details.

Figure 11.3 — Event handling, compared: JavaFX's setOnAction(...) and Tkinter's command=... both bind a callback the toolkit invokes only in response to a real user action, but differ in what the callback receives and who owns the event loop.

LibraryController.java (excerpt)

```
public class LibraryController {
    private final Library library;
    private final LibraryView view;

    public LibraryController(Library library, LibraryView view) {
        this.library = library;
        this.view = view;
        wireEvents();
        refreshView();
    }

    private void wireEvents() {
        view.getAddButton().setOnAction(e -> handleAddBook());
        view.getBorrowButton().setOnAction(e -> handleBorrowSelected());
        view.getReturnButton().setOnAction(e -> handleReturnSelected());
        view.getRemoveButton().setOnAction(e -> handleRemoveSelected());
    }

    private void handleBorrowSelected() {
        LibraryItem selected = view.getItemList()
            .getSelectionModel().getSelectedItem();
        if (selected == null) {
            view.setStatus("Borrow failed: no item selected.");
            return;
        }
        boolean ok = selected.borrow();
        view.setStatus(ok
            ? "Borrowed: " + selected.getTitle()
            : "Borrow refused: " + selected.getTitle() + " is unavailable "
              + "(already on loan, or a reference-only item).");
        refreshView();
    }

    private void refreshView() {
        view.refresh(library.getAllItems());
    }
}
```

A layout that is too narrow can silently truncate a control's own label

The first version of LibraryView built in this chapter packed all three action buttons into a single HBox inside a 280-pixel-wide side panel -- the code compiled cleanly, ran without error, and every button fired its correct handler when clicked. Only a screenshot from the headless verification run (Section 11.7) revealed the real defect: "Borrow Selected" rendered as "Borro…", each label cut off because the HBox had nowhere near enough horizontal room for three full-width buttons. This is not a logic bug -- compareTo(), borrow(), and every handler above were already correct -- it is a LAYOUT defect, invisible to javac and invisible to a code read-through, and it was fixed by stacking the buttons in a VBox instead, with borrowButton.setMaxWidth(Double.MAX_VALUE) (and the same for the other two) so each button stretches to the full width of its container. A GUI chapter's bugs are not always in the logic; some are only visible in a picture of the running window.

11.6 The Application Entry Point: LibraryApp

LibraryApp extends javafx.application.Application, JavaFX's required base class for any GUI program's entry point. Its start(Stage stage) method runs once: it builds the Model (a Library pre-populated with the same four items this course has used since Chapter 7-10), constructs the View and the Controller that connects it to that Model, and shows the Stage (JavaFX's word for a top-level window). After stage.show() returns, control passes to JavaFX's own event loop, and nothing else in start() runs during an ordinary interactive session -- every subsequent line of behavior comes from a Controller handler responding to a real click.

LibraryApp.java (excerpt)

```
public class LibraryApp extends Application {
    @Override
    public void start(Stage stage) throws IOException {
        Library library = new Library();
        library.addItem(new Book("Clean Code", "Robert C. Martin",
            "9780132350884", 2008));
        library.addItem(new DVD("The Imitation Game",
            "9900000000001", 114));
        library.addItem(new ReferenceBook("Encyclopedia of Computer Science",
            "Ralston & Reilly", "9780123456789", 2003));
        library.addItem(new Book("Algorithms", "Robert Sedgewick",
            "9780321573513", 2011));

        LibraryView view = new LibraryView();
        LibraryController controller = new LibraryController(library, view);

        Scene scene = view.buildScene();
        stage.setTitle("Pustaka -- Library System (Chapter 11)");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Application.launch() and JavaFX's own event loop

launch(args) is what actually starts JavaFX: it initializes the JavaFX runtime, creates the Stage that gets passed to start(), and then begins the toolkit's own rendering-and-event thread, which is what waits for clicks and dispatches them to whichever handler was wired with setOnAction(...). main() calling launch(args) and then doing nothing else is normal and expected -- there is no line of code in this program that runs the equivalent of a while(true) loop; JavaFX supplies that loop itself, and the program only ever reacts.

11.7 Putting It Together: Verifying a GUI Actually Runs

A console chapter's driver produces a text transcript that is trivially reproducible: run it again, get the identical System.out lines. A GUI has no equivalent text output by default, so this chapter's LibraryApp additionally supports a scripted verification path, gated behind the JVM system property -

Dpustaka.verify=true (read via Boolean.getBoolean("pustaka.verify")), that a normal interactive run never takes. When that property is set, start() calls runScriptedVerification(), which saves a screenshot of the freshly opened window, selects the first list row PROGRAMMATICALLY (view.getItemList().getSelectionModel().select(0)), then calls view.getBorrowButton().fire() -- JavaFX's own API for firing a button exactly as a real mouse click would, running through the identical setOnAction(...) lambda a click dispatches to -- and saves a second screenshot afterward.

LibraryApp.java -- runScriptedVerification() (excerpt)

```
private void runScriptedVerification(LibraryView view, Scene scene)
        throws IOException {
    saveSnapshot(scene, "gui_screenshot_1_initial.png");

    view.getItemList().getSelectionModel().select(0);
    view.getBorrowButton().fire();   // exact same code path as a real mouse
click
    System.out.println("status now reads: \"" + view.getStatusLabel().getText()
+ "\"");

    saveSnapshot(scene, "gui_screenshot_2_after_borrow.png");
    Platform.exit();
}

private void saveSnapshot(Scene scene, String fileName) throws IOException {
    WritableImage image = scene.snapshot(null);
    ImageIO.write(SwingFXUtils.fromFXImage(image, null), "png", new
File(fileName));
}
```

Button.fire() does not simulate or describe a click; it runs the identical setOnAction(...) callback JavaFX itself calls when the mouse actually clicks that button, so the resulting screenshots and status text are genuine proof that LibraryController's handler chain -- read the View's selection, mutate the Model via selected.borrow(), tell the View to refresh -- works correctly end to end, not a hand-written description of what the author expects it to do. Both screenshots are reproduced in Appendix 11.A, alongside the complete text log this run produced.

11.8 Compiling and Running Your Program

JavaFX ships as a separate set of modules from the JDK itself (this course's environment uses OpenJFX 11.0.11), so both compiling and running need two extra flags beyond every prior chapter's plain javac/java: --module-path pointing at the JavaFX jars, and --add-modules naming the specific modules this chapter's code uses (javafx.controls for the UI controls, javafx.swing for SwingFXUtils, used only by the verification path's screenshot code).

```
$ javac --module-path /usr/share/openjfx/lib --add-modules
javafx.controls,javafx.swing \
        LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java \
        ItemNotFoundException.java Pair.java LibraryUtils.java Library.java \
        LibraryView.java LibraryController.java LibraryApp.java
$ java --module-path /usr/share/openjfx/lib --add-modules
javafx.controls,javafx.swing \
       LibraryApp

# scripted verification run (produces the two screenshots in Appendix 11.A):
$ java --module-path /usr/share/openjfx/lib --add-modules
javafx.controls,javafx.swing \
       -Dpustaka.verify=true LibraryApp
```

Verifying a real GUI without a physical screen

This chapter's code was compiled and run in a sandboxed environment with no physical display attached, using Xvfb, a virtual X11 display server: xvfb-run -a java ... gives JavaFX a display to render into that exists only in memory, letting the program run, click through its real event handlers, and produce genuine screenshots without a monitor. This is disclosed here rather than glossed over: the verification is completely real -- the same JavaFX code, the same event dispatch, the same rendering pipeline a desktop run would use -- only the display itself is virtual, not simulated.

11.9 OOP in Practice: Profitina

About this box

As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author -- describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's internal admin dashboards follow the same MVC split this chapter teaches: a backend service layer (the Model, in spirit) that knows nothing about how any dashboard renders a chart, a rendering layer (the View) that only knows how to draw whatever data it is handed, and a thin coordinating layer (the Controller) that reads what an operator clicks, asks the backend service to do something, and tells the rendering layer to redraw. The same discipline this chapter enforces at toy scale -- the Model never imports anything about the View -- is what lets Profitina's backend be tested, and even reused, entirely independently of any particular dashboard built on top of it.

11.10 Chapter Summary and Key Takeaways

- The Model-View-Controller pattern splits a GUI program into a Model (data and rules, no GUI knowledge), a View (visible controls, zero business logic), and a Controller (the only class allowed to depend on both).
- Chapters 5-10's Library and LibraryItem hierarchy needed exactly one new method (getAllItems(), a defensive-copy accessor) to become usable from a GUI View -- every borrow()/returnItem()/removeItemOrThrow() call is reused completely unchanged.

- Event-driven programming inverts a console program's control flow: JavaFX's own event loop, not the programmer's code, decides when a handler runs, calling back into exactly the lambda wired with setOnAction(...) for whichever control the user just interacted with.
- Every Controller handler follows the same three-step shape: read what the View currently shows, tell the Model to do something, then tell the View to refresh -- the same shape this chapter's Python-track Controller (Section 11.5 of that book) uses with Tkinter's command= callbacks instead.
- Button.fire() runs the identical event-handler code path a real mouse click dispatches to, making a scripted, headless verification run genuine proof the GUI works, not a description of expected behavior -- and that same verification run is what caught this chapter's one real defect, a layout too narrow to show three full button labels.

A View that never imports the Model's business rules, and a Model that never imports anything about the View, is not an abstract ideal in this chapter -- it is the reason LibraryView's only unplanned defect was a pixel-width problem, and never a single wrong dollar amount, a single wrong borrow decision, or a single stale list of items.

11.11 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. LibraryController.handleRemoveSelected() catches ItemNotFoundException even though the ISBN it passes to removeItemOrThrow() came directly from a LibraryItem already sitting in the View's own list. Explain why that catch block is still required for the code to compile, and why the branch inside it should never actually run in practice.
2. Suppose LibraryView exposed library directly (a public field or getter) instead of only exposing its own controls. Describe one change LibraryController could then make that would violate the MVC boundary this chapter establishes, and explain what would go wrong if two different Views both tried to do this.
3. view.getBorrowButton().fire() is used in this chapter's verification run instead of directly calling library.findByTitle(...).borrow() from the verification code. Explain what a direct Model call like that would fail to prove that fire() does prove.
4. The button-label-truncation defect in Section 11.5 was invisible to javac and invisible to a plain code read-through. Name one other kind of GUI defect (not layout-related) that would also compile and run without error, yet only be caught by looking at an actual screenshot.
5. Rewrite handleBorrowSelected() so that, instead of calling selected.borrow() directly, it first checks selected.isOnLoan() and sets a status message of "Already on loan." before ever calling borrow() at all. Would this change the Model's public interface? Explain your answer.

Appendix 11.A — Execution Verification Log

The complete LibraryItem.java, Renewable.java, Book.java, DVD.java, ReferenceBook.java, ItemNotFoundException.java, Pair.java, LibraryUtils.java, Library.java, LibraryView.java, LibraryController.java, and LibraryApp.java files (Code/Java/Chapter11/, provided alongside this book) were compiled and executed under Xvfb on OpenJDK 21.0.10 with OpenJFX 11.0.11 before this chapter was written, using the -Dpustaka.verify=true scripted path described in Section 11.7. Both

screenshots below are genuine, unedited output from that run -- not mockups -- and the text log beneath them is reproduced verbatim (Figures 11.4 and 11.5).

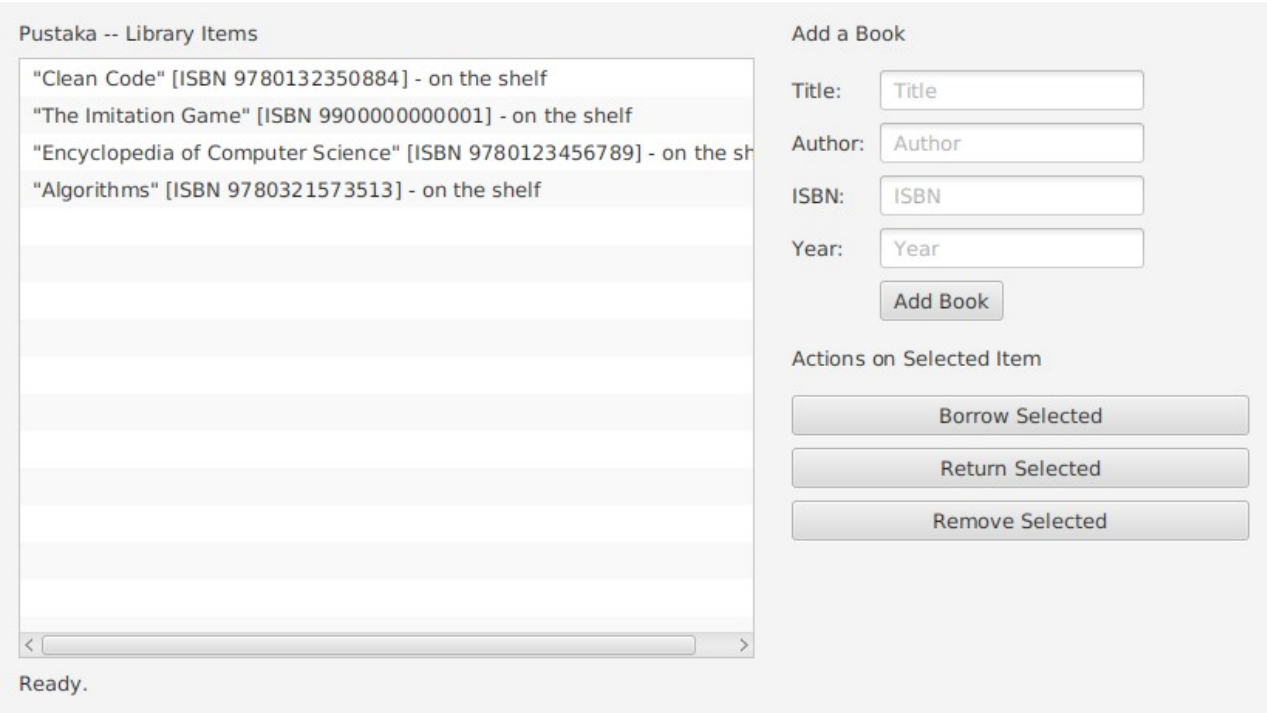


Figure 11.4 — Screenshot 1: the window immediately after opening. All four items show "on the shelf"; nothing is selected yet.

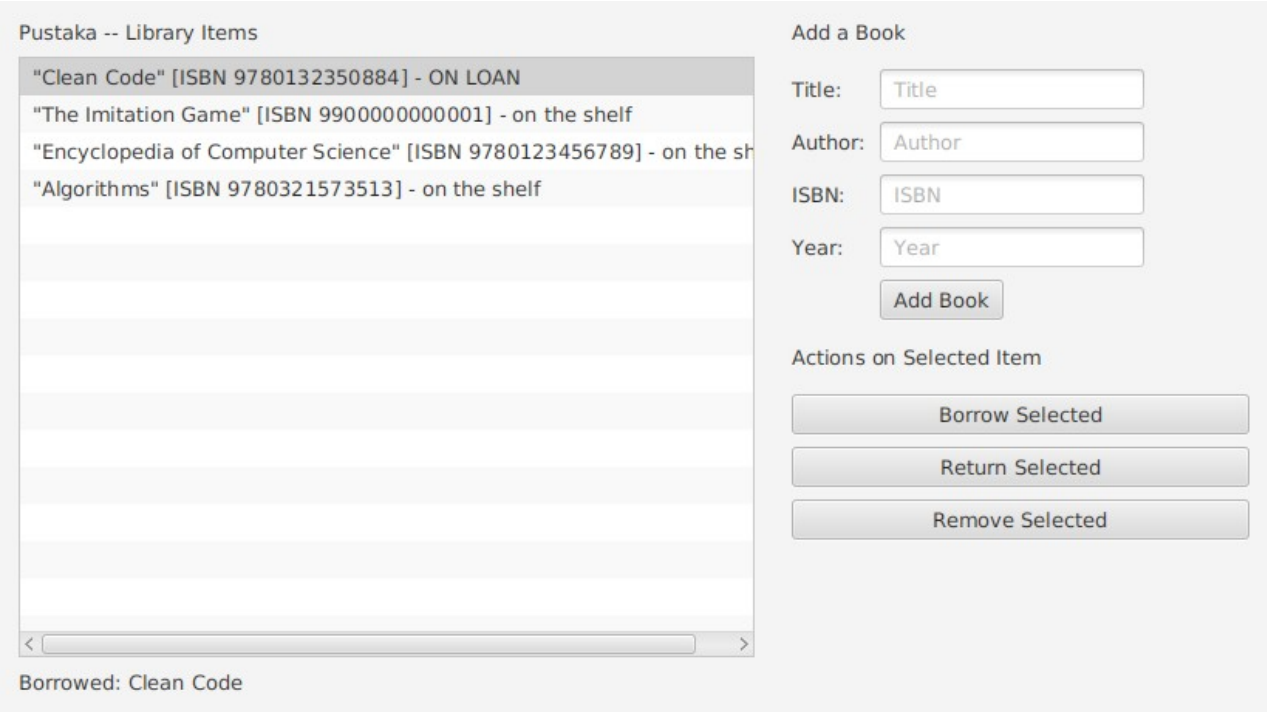


Figure 11.5 — Screenshot 2: after the scripted run selected row 0 and fired Borrow Selected. "Clean Code" is highlighted and now reads "ON LOAN"; the status label reads "Borrowed: Clean Code".

```
$ javac --module-path /usr/share/openjfx/lib --add-modules
javafx.controls,javafx.swing *.java
(no errors)

$ xvfb-run -a java --module-path /usr/share/openjfx/lib --add-modules
javafx.controls,javafx.swing \
           -Dpustaka.verify=true LibraryApp
--- Chapter 11 GUI verification run ---
Snapshot 1 saved: initial window, 4 items loaded, nothing selected.
Selected list row 0 (programmatically, same selection a click would make).
Fired Borrow Selected button -> status now reads: "Borrowed: Clean Code"
Snapshot 2 saved: after borrowing the selected item.
```

References

[1] Oracle Corporation. "JavaFX Overview." JavaFX Documentation Project. https://openjfx.io/openjfx-docs/ (accessed August 2026).

[2] Oracle Corporation. "Handling JavaFX Events." JavaFX Documentation Project. https://openjfx.io/javadoc/21/javafx.graphics/javafx/event/package-summary.html (accessed August 2026).

[3] G. E. Krasner and S. T. Pope. "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System." Journal of Object-Oriented Programming, 1(3), 1988.

[4] Oracle Corporation. "Scene.snapshot(SnapshotParameters, WritableImage)." JavaFX API Documentation. https://openjfx.io/javadoc/21/javafx.graphics/javafx/scene/Scene.html (accessed August 2026).

CHAPTER 12 · EXERCISE CHAPTER

Practice Problems: Quiz 2 — Reviewing Lectures 1 Through 11

No new material here — eight problems spanning everything from encapsulation to Model-View-Controller design, reviewing every lecture since Lecture 1 before Quiz 2.

Learning Objectives — after working through this chapter you will be able to:

(1) Design a small encapsulated class that stores a couple of values and derives others from them on demand. (2) Write control-flow logic that behaves correctly at every boundary value, not just the obvious cases. (3) Build a formatted string efficiently from a list, using StringBuilder rather than repeated concatenation. (4) Define and raise a custom exception that leaves an object's state untouched on failure, and read and write a plain-text log file safely. (5) Design an abstract class where a concrete method is built on top of an abstract one, and extend a class with true inheritance that calls the parent implementation via super and dispatches polymorphically over a mixed list. (6) Write a generic method bounded to Comparable, build an index using a Map, and sketch a Model-View-Controller design where each class has a single, clearly bounded responsibility.

12.1 Recap: Lectures 1–11 in Brief

Lectures 1 through 8 built the language's core toolkit and were already reviewed in full by Chapter 8's Midterm: the shift from procedural to object-oriented thinking (Lecture 1), class anatomy and encapsulation (Lecture 2), primitives and control flow (Lecture 3), Quiz 1 (Lecture 4), arrays, Lists, and StringBuilder (Lecture 5), exceptions and file I/O (Lecture 6), interfaces and abstract classes (Lecture 7), and the Midterm itself (Lecture 8). Lecture 9 built directly on Lecture 7's abstract classes and interfaces with true inheritance — extends, super, method overriding, and dynamic dispatch resolving polymorphically at runtime rather than at compile time. Lecture 10 added Java's Generics and the Collections Framework, letting Pustaka's LibraryItem hierarchy be stored, sorted, and looked up in a type-safe List, Set, or Map instead of a raw array. Lecture 11 took Library out of the console entirely, building LibraryView and LibraryController on top of JavaFX's own event loop under the Model-View-Controller pattern — the same pattern Problem 8 below asks you to sketch for a much smaller application (Figure 12.1).

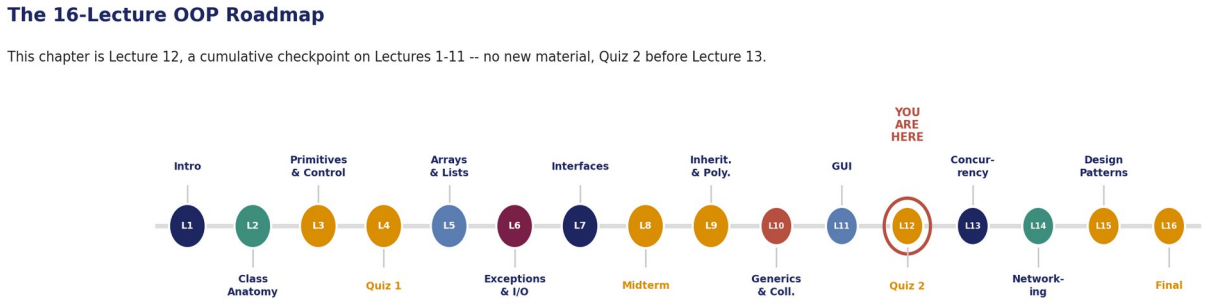


Figure 12.1 — This chapter sits at Lecture 12 in the sixteen-lecture OOP roadmap: a checkpoint, not a new topic, Quiz 2 before Lecture 13.

12.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a Bab Latihan: a practice chapter, not a new-content chapter. Every problem below is anchored to specific material from one of the eight lectures since Lecture 1 that taught new content (see Figure 12.2). Each problem comes with a HINT — a nudge toward the right way of thinking — but deliberately NOT a full worked solution or source code. These eight problems are standalone, independent of the Pustaka case study that runs through the ordinary chapters — exactly like Quiz 1's problems in Chapter 4 and the Midterm's problems in Chapter 8.

Where Each Practice Problem Comes From

Every problem in Section 12.3 traces back to one of the eight lectures since Lecture 1 that taught new material.

#	Problem	Traces back to
1	Rectangle Class	L2 -- fields, constructor, encapsulation
2	BMI Category	L3 -- control flow, boundary conditions
3	CSV Row Formatter	L5 -- StringBuilder, arrays and lists
4	Negative Balance Guard	L6 -- a custom checked exception, a transaction log file
5	Shape Hierarchy	L7 -- abstract class, a concrete method built on an abstract one
6	Vehicle & Car	L9 -- inheritance, polymorphism, super.describe()
7	Generic maxOf	L10 -- bounded generics over Comparable, a Map-based index
8	Tiny MVC Counter	L11 -- Model-View-Controller, event-driven design

Figure 12.2 — Every problem in Section 12.3 traces back to one of the eight lectures since Lecture 1 that taught new material.

Before you start

Try each problem for real in a fresh .java project before reading its hint. If you get stuck, read only the hint — not a solution — and try again. This is exactly the discipline Section 12.4's Quiz 2 will reward: the assessment is open-book, but that is not a substitute for your own reasoning.

12.3 Practice Problems

12.3.1 Fields, Constructors & Encapsulation Review

Problem 1 *[Easy]*. Define a class Q121Rectangle(double width, double height), storing only the two dimensions, with getWidth(), getHeight(), getArea() (width * height), and getPerimeter() (2 * (width + height)).

Hint

Store only the width and height fields. Both getArea() and getPerimeter() should compute their result on demand from those two fields, rather than storing area and perimeter as fields that could fall out of sync if width or height were ever changed — exactly the same discipline as Chapter 8's Q81Temperature.

12.3.2 Primitives & Control Flow Review

Problem 2 *[Easy]*. Write a class Q122BmiCalculator with a static method String bmiCategory(double weightKg, double heightM) that computes bmi = weightKg / (heightM * heightM) and returns "Underweight" for bmi < 18.5, "Normal" for bmi < 25.0, "Overweight" for bmi < 30.0, and "Obese" otherwise. Test at 18.5, 18.49, 25.0, 24.99, 30.0, and 29.99.

> **Hint**
>
> Order your comparisons from lowest cutoff to highest this time, since the categories are phrased with < rather than >= — an if/else-if chain that returns as soon as one condition matches. The 18.49/24.99/29.99 cases exist specifically to catch a < versus <= mistake at exactly the cutoff value, which the round numbers alone would never reveal.

12.3.3 Arrays, Lists & StringBuilder

Problem 3 *[Medium]*. Write a static method String formatCsvRow(String[] fields) that joins fields with commas into one CSV line, except that any field containing a comma must itself be wrapped in double quotes first (a simplified version of the real CSV quoting rule). Example: {"Ann", "Budi, Jr.", "30"} becomes Ann,"Budi, Jr.",30. Build the result with a StringBuilder rather than repeated String concatenation.

> **Hint**
>
> Loop with an index so you know whether you are on the last field (no trailing comma to append). For each field, check field.contains(",") first — if true, wrap that one field in double-quote characters before appending it, otherwise append it unchanged.

12.3.4 Exception Handling & File I/O

Problem 4 *[Medium]*. Define a custom checked exception class NegativeBalanceException, and a class Q124Account(double openingBalance) with a method void withdraw(double amount) throws NegativeBalanceException, IOException that throws a NegativeBalanceException (whose message includes both the requested amount and the current balance) if amount exceeds the current balance, and otherwise deducts amount from the balance and appends one line "WITHDRAW,amount,balanceAfter" to a transaction log file.

> **Hint**
>
> Check the insufficient-funds condition and throw BEFORE touching the balance field or opening the file — a failed withdrawal must leave both completely untouched. Once the check passes, update the balance field first, then use try-with-resources with a BufferedWriter opened in append mode to write the log line, exactly the pattern Chapter 6's saveToFile used for Pustaka and Chapter 8's Q86 log file used for scores.

12.3.5 Interfaces & Abstract Classes

Problem 5 *[Medium]*. Define an abstract class Q125Shape with an abstract method double area(), and a CONCRETE method String describe() built entirely from area() (e.g. returning something like "This shape covers 28.27 square units."). Then define two concrete subclasses: Q125Circle(double radius) and Q125Square(double side).

> **Hint**
>
> describe() belongs on the abstract class itself, not on either subclass — it is already fully implementable in terms of whatever area() each subclass eventually provides, exactly like Chapter 8's Q87Employee.annualPay() being a concrete method built on top of the abstract monthlyPay().

12.3.6 Inheritance & Polymorphism

Problem 6 *[Medium]*. Define a class Q126Vehicle(String make, String model) with a method String describe() returning "make model". Then define a subclass Q126Car(String make, String model, int doorCount) that OVERRIDES describe() to call super.describe() and append the door count, e.g. "Toyota Corolla (4 doors)". Finally, build a List<Q126Vehicle> containing a mix of plain Q126Vehicle and Q126Car objects, and call describe() on each in a loop.

> **Hint**
>
> Q126Car.describe() must call super.describe() rather than reformatting make and model itself — that is inheritance reusing the parent's logic, not duplicating it. The loop over List<Q126Vehicle> is polymorphic dispatch: even though the reference type is Q126Vehicle, each call to describe() resolves at runtime to whichever class the actual object belongs to.

12.3.7 Generics & the Collections Framework

Problem 7 *[Hard]*. Write a generic static method <T extends Comparable<T>> T maxOf(List<T> items) that returns the largest element of items (throwing IllegalArgumentException for an empty list). Then write a static method Map<Character, List<String>> groupByFirstLetter(List<String> words) that groups words into a Map keyed by their (uppercased) first letter.

> **Hint**
>
> The <T extends Comparable<T>> bound is what lets maxOf call item.compareTo(other) without an unchecked cast — without the bound, T could be any type at all, including one with no ordering. For groupByFirstLetter, use Map.computeIfAbsent(key, k -> new ArrayList<>()).add(word) so you never need a separate containsKey check before adding to a group's list.

12.3.8 GUI & MVC

Problem 8 *[Hard]*. Design a tiny MVC counter application: a Model class Q128CounterModel with increment(), decrement(), and getValue(); a View class that shows the current value in a label and offers a "+" button and a "-" button; and a Controller class that is the ONLY class holding a reference to both the Model and the View, wiring each button to the matching Model method and then refreshing the View's label. Sketch the three classes' fields and method signatures — you do not need a working GUI to answer this problem.

Hint

Exactly as in Chapter 11's LibraryView/LibraryController: the Model must have zero import of your GUI toolkit, the View must have zero business logic (no incrementing happens inside the View), and only the Controller is allowed to import and depend on both. If any button's click handler modifies Q128CounterModel's internal count directly instead of calling increment()/decrement(), that view has broken encapsulation exactly as Chapter 11 warned against.

12.4 Quiz 2 Format: Open-Book, Individual, Timed

Quiz 2 is an open-book, individual, timed take-home assessment (roughly 120 minutes) covering exactly the eight kinds of problems reviewed in this chapter — a cumulative checkpoint on everything since Lecture 1, not only the material since the Midterm. Grading is per-problem, not all-or-nothing: partial credit is given for a class that compiles and handles the general case correctly even if one edge case is missed.

Open-book means references, not collaborators

You may consult the textbook, your own notes, and lecture slides while answering. You may NOT collaborate with classmates or submit code that is not genuinely your own — an open-book timed quiz tests whether you can APPLY what you have studied, unsupervised, under a time limit.

Criterion	What it looks for	Weight
Correctness of output	Does the class, method, or function produce the exact expected result for every given test case, including boundary values?	45%
Class, interface & generic design	Are fields private, is state reachable only through methods, and are the abstract-class split (Problem 5) and the generic bound (Problem 7) used correctly?	25%
Exception, I/O & MVC discipline	Does Problem 4's exception carry a useful message and leave the account's state untouched on failure, does its file I/O use try-with-resources correctly, and does Problem 8's sketch keep Model, View, and Controller cleanly separated?	15%
Code clarity & compiles cleanly	Is the code easy to read, sensibly named, and free of dead code — and does it compile with javac with zero errors?	15%

12.5 OOP in Practice: Profitina

About this box

As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

This chapter's eight problems, taken together, are close to a miniature version of what a single feature at Profitina actually requires: a small encapsulated class or two (Problem 1), careful boundary-tested logic (Problem 2), efficient text handling for a report or API response (Problem 3), an operation that must fail safely without corrupting existing state, logged durably to disk (Problem 4), a shared contract implemented by several concrete shapes of data (Problem 5), a hierarchy that reuses a parent's behavior rather than duplicating it (Problem 6), a generic utility usable across many unrelated data types plus a Map-based index for fast lookup (Problem 7), and a Model-View-Controller split that keeps a dashboard's rendering logic ignorant of the backend service feeding it (Problem 8, the same

split Profitina's own admin dashboards use). A Quiz that only tested one of these skills would miss how often real features need several of them working together in the same small piece of code.

12.6 Chapter Summary

- This chapter introduced no new material — it is a cumulative checkpoint covering Lectures 1 through 11.
- Eight standalone problems span the full range reviewed: encapsulation (L2), control-flow boundaries (L3), StringBuilder (L5), custom exceptions and file I/O (L6), abstract classes (L7), inheritance and polymorphism (L9), generics and the Collections Framework (L10), and Model-View-Controller design (L11).
- Each problem includes a hint, not a solution — working through the reasoning and the code yourself is the point.
- Quiz 2 is an open-book, individual, timed take-home assessment: references are allowed, but the code must be your own, and correctness on every given test case (including boundary values) is the largest share of the grade.

A class that compiles is not the same as a class that is correct — and a design that looks clean on the page is not the same as one that actually keeps its Model, View, and Controller from depending on each other.

Appendix 12.A — Self-Check Checklist (Non-Graded)

Before submitting any answer — in this chapter's problems or in Quiz 2 itself — run it through this checklist. It costs a few minutes and catches most of the mistakes graders see most often.

- Does every class and method compile with javac with zero errors and zero warnings?
- Is every field private, with every value the caller needs exposed only through a method?
- Did I test Problem 2's exact boundary values (18.49, 24.99, 29.99), not just the round numbers that a < versus <= bug could hide behind?
- Does Problem 4's withdraw() leave the balance field and the log file completely untouched when it throws NegativeBalanceException?
- Are Problem 5's abstract method and concrete method on the SAME abstract class, with the concrete method calling the abstract one rather than duplicating logic in each subclass?
- Does Problem 6's Q126Car.describe() call super.describe() rather than reformatting make and model itself?
- Does Problem 7's maxOf() compile only for types that implement Comparable, and does groupByFirstLetter() avoid a separate containsKey check before adding to a group?
- Did I reread my own code as if I were a skeptical grader looking for the one input that breaks it?

References

[1] This exercise chapter's assessment format (open-book, individual, timed) is modeled on this course's real Quiz 2 (EF234302 Object-Oriented Programming), a group project building a T9 predictive-text system. That project's first three parts center on a custom multi-branch recursive tree, binary search over a sorted array, and multi-valued Map lookups — the same class of Data Structures material this rebuild's real Midterm (see Chapter 8's Reference [1]) already placed outside OOP's own scope, so those parts were not reused here either. Its fourth part, however, builds a phone-keypad GUI using the Model-View-Controller pattern — the same pattern this rebuild actually taught in Chapter 11 — so Problem 8 above draws its GUI/MVC theme directly from that part, while the other seven problems were designed fresh to test the seven other lecture areas this Quiz 2 needed to cover.

[2] Oracle Corporation. "Interfaces and Inheritance." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/ (accessed August 2026).

[3] Oracle Corporation. "Lesson: Generics." The Java Tutorials. https://docs.oracle.com/javase/tutorial/extra/generics/ (accessed August 2026).

CHAPTER 13

Concurrency: Threads, Race Conditions, and Synchronization

Every chapter so far has run one piece of code at a time, in one order, decided entirely by the program itself. Real systems -- including Pustaka's own public kiosks -- don't get that luxury: several patrons can act on the library at the exact same moment. This chapter shows, with a real bug you can reproduce yourself, why that changes everything about how shared state must be written (Figure 13.1).

Learning Objectives — after this chapter you will be able to:

(1) Explain why an operation as simple as count++ is not atomic, and describe concretely how two threads can interleave to lose an update. (2) Create and start multiple Threads to run work concurrently, and join() them to wait for every one to finish. (3) Use the synchronized keyword to enforce mutual exclusion on shared state, and explain exactly what "acquiring a lock" blocks other threads from doing. (4) Use an ExecutorService thread pool to run many short tasks without manually managing one Thread object per task. (5) Read a real captured program run and distinguish a race condition that is merely theoretically possible from one that has actually been observed, with an exact number attached.

13.1 Recap: Pustaka So Far

Chapter 9 generalized Library to hold any LibraryItem polymorphically. Chapter 10 added Generics and the Collections Framework. Chapter 11 gave Library a real windowed interface built on the Model-View-Controller pattern. Chapter 12 was a checkpoint, not new material. Every one of those chapters, and every chapter before them, ran on a single thread: one call stack, one order of execution, entirely predictable from reading the source top to bottom. This chapter is the first time that stops being true.

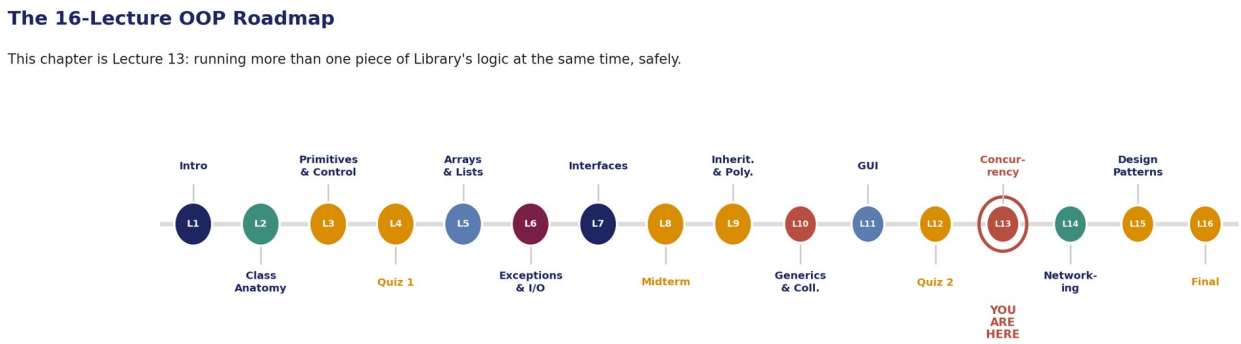


Figure 13.1 — The 16-lecture OOP roadmap. This chapter is Lecture 13: running more than one piece of Library's logic at the same time, safely.

13.2 Why Concurrency: Threads and Shared State

A Thread is an independent path of execution inside the same running program, sharing the same objects and memory as every other thread the program creates. Pustaka's real deployment plan (Chapter 11's GUI notwithstanding) imagines several public kiosks, each letting a different patron borrow items at the same moment -- and if those kiosks are each backed by their own thread, all of them can end up calling methods on the very same shared object at the very same instant. That is the

situation this chapter's new class, BorrowCounter, is built to expose: a single shared count of how many items have been borrowed across the whole library today, updated from multiple threads at once.

BorrowCounter.java -- a minimal shared-state contract

```
public interface BorrowCounter {
    void increment();
    int getCount();
}
```

Two classes implement this interface. One of them has a bug that is invisible from reading the source code alone -- it only shows up when real threads actually run it.

13.3 The Race Condition: A Real, Reproducible Bug

count++ (and count += 1) looks like a single, indivisible operation in Java source code. It is not. The JVM performs it as three separate steps: read the current value of count, add one to it, and write the new value back. Nothing prevents two threads from interleaving those three steps -- and when they do, an increment is silently lost (Figure 13.2).

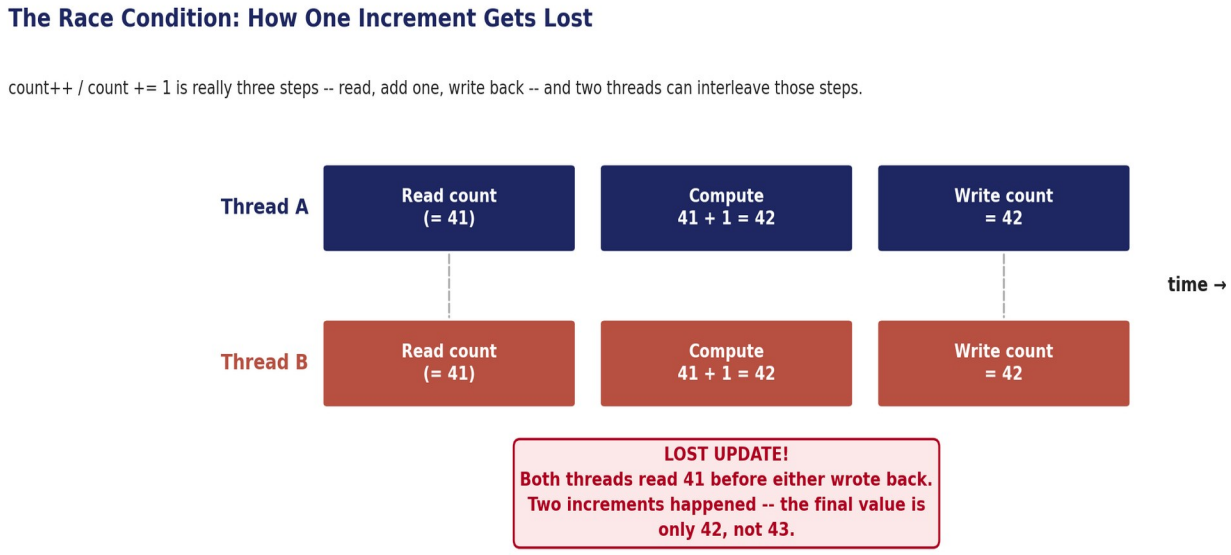


Figure 13.2 — Two threads both read count as 41 before either writes back; both compute 42 and write 42. One of the two increments never happened, and nothing in the program raised an error to say so.

UnsafeBorrowCounter.java -- the naive, buggy implementation

```
public class UnsafeBorrowCounter implements BorrowCounter {
    private int count = 0;

    @Override
    public void increment() {
        count++;
    }

    @Override
    public int getCount() {
        return count;
    }
}
```

count++ compiles cleanly, runs without error, and is still wrong

There is no exception, no warning, no crash -- UnsafeBorrowCounter behaves perfectly under a single thread, and javac has absolutely nothing to say about it. The bug only exists in the gap between reading and writing, and that gap is only ever exposed by running real concurrent threads against it, which is exactly what Section 13.7's driver program does.

13.4 Synchronization: Mutual Exclusion with synchronized

Marking a method synchronized tells the JVM to acquire the calling object's own intrinsic (monitor) lock before running the method body, and to release it only when the method returns. While one thread holds that lock, every other thread that calls a synchronized method on the SAME object blocks -- it simply waits -- until the lock is released. That turns "read, add one, write back" into an operation that behaves atomically from every other thread's point of view, even though it is still three JVM instructions underneath.

SafeBorrowCounter.java -- the fixed implementation

```
public class SafeBorrowCounter implements BorrowCounter {
    private int count = 0;

    @Override
    public synchronized void increment() {
        count++;
    }

    @Override
    public synchronized int getCount() {
        return count;
    }
}
```

getCount() needs synchronized too, not just increment()

Synchronizing only increment() would stop two increments from interleaving with EACH OTHER, but a thread calling getCount() while another thread is mid-increment could still read a value that is about to change -- a subtler, still-real hazard called a visibility problem. Synchronizing both methods on the same object's lock closes both gaps at once.

13.5 A Thread Pool: ExecutorService

Creating and starting one Thread object per task, as Section 13.7's first two demos do, does not scale past a handful of tasks -- each Thread carries real memory and OS scheduling overhead. ExecutorService, from java.util.concurrent, manages a fixed, reusable pool of worker threads instead: submit() hands it a task, and whichever pool thread is free next picks it up, with no new Thread object created per task (Figure 13.3).

Mutual Exclusion, Compared: Two Ways to Say the Same Thing

Both languages let you mark "only one thread at a time may run this" -- the mechanism and the syntax differ.

	Java	Python
Declaring mutual exclusion	synchronized void increment() { ... } a keyword on the method itself	with self._lock: self._count += 1 an explicit threading.Lock object
What is held while inside	The object's own intrinsic (monitor) lock -- built into every object	Whatever Lock object you created and chose to acquire
A thread pool for many tasks	ExecutorService + Executors.newFixed-ThreadPool(n)	concurrent.futures.ThreadPoolExecutor(max_workers=n)

Neither mechanism removes the underlying hazard by magic -- both work by making every OTHER thread wait its turn at the one line of code that touches the shared value, so "read, add one, write back" becomes effectively one step from every other thread's point of view.

Figure 13.3 — Mutual exclusion and thread pools, compared across both tracks this course uses.

Submitting work to a fixed thread pool

```
ExecutorService pool = Executors.newFixedThreadPool(4);
for (int i = 0; i < KIOSK_COUNT; i++) {
    pool.submit(new BorrowWorker(counter, ITERATIONS));
}
pool.shutdown();
pool.awaitTermination(1, TimeUnit.MINUTES);
```

shutdown() does not stop already-submitted work

pool.shutdown() tells the pool to accept no NEW tasks and to terminate its worker threads once every already-submitted task finishes -- it is not an abrupt stop. awaitTermination() then blocks the calling thread until that finishing-up is complete (or the given timeout elapses), which is what lets the driver program safely read the final count immediately afterward.

13.6 Concurrency and Library: BorrowCounter as a Case Study

BorrowCounter is deliberately a small, self-contained class rather than a change to Library itself -- exactly the same decision Chapter 11 made when it added getAllItems() rather than rewriting Library's internals for a GUI. The lesson this chapter teaches (shared mutable state needs explicit synchronization the moment more than one thread can reach it) applies to any of Library's own fields

exactly as much as it applies to BorrowCounter's single int; BorrowCounter simply isolates that lesson from every other concern this course has already covered, so Section 13.7 can demonstrate it in isolation.

BorrowWorker.java -- simulates one patron kiosk

```
public class BorrowWorker implements Runnable {
    private final BorrowCounter counter;
    private final int iterations;

    public BorrowWorker(BorrowCounter counter, int iterations) {
        this.counter = counter;
        this.iterations = iterations;
    }

    @Override
    public void run() {
        for (int i = 0; i < iterations; i++) {
            counter.increment();
        }
    }
}
```

13.7 The Driver Program: Three Demos, Back to Back

ConcurrencyDemo.java simulates 8 patron kiosks, each calling increment() 200,000 times -- so the mathematically correct final count is always 8 * 200,000 = 1,600,000, regardless of which BorrowCounter is used. Demo 1 runs that workload against UnsafeBorrowCounter with 8 manually created Threads; Demo 2 runs the identical workload against SafeBorrowCounter the same way; Demo 3 runs it a third time against SafeBorrowCounter, but through a 4-thread ExecutorService pool instead of 8 manually managed Threads.

ConcurrencyDemo.java -- running the unsafe demo with manual threads (abridged)

```
private static int runWithManualThreads(BorrowCounter counter) throws
InterruptedException {
    Thread[] kiosks = new Thread[KIOSK_COUNT];
    for (int i = 0; i < KIOSK_COUNT; i++) {
        kiosks[i] = new Thread(new BorrowWorker(counter, ITERATIONS));
    }
    for (Thread kiosk : kiosks) { kiosk.start(); }
    for (Thread kiosk : kiosks) { kiosk.join(); }
    return counter.getCount();
}
```

start() begins a thread; calling run() directly does not

kiosk.start() asks the JVM to begin a genuinely new thread of execution, which then calls run() on its own. Calling kiosk.run() directly would just execute the loop on the CURRENT thread, sequentially, with no concurrency at all -- a common, easy mistake that produces code that compiles and even produces the right number, for the wrong reason.

13.8 Compiling and Running Your Program

javac every .java file in the Chapter 13 folder, then java ConcurrencyDemo.

```
$ javac BorrowCounter.java UnsafeBorrowCounter.java SafeBorrowCounter.java \
      BorrowWorker.java ConcurrencyDemo.java
$ java ConcurrencyDemo
```

13.9 OOP in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author -- describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's backend serves many concurrent requests at once, and several of its own counters -- rate limits, in-flight request tallies, cache hit/miss statistics -- face exactly this chapter's hazard: a naive shared counter updated from multiple request-handling threads without synchronization would silently under-count under real production load, the same way UnsafeBorrowCounter under-counts here. The fix in production code is the same idea this chapter teaches at a much smaller scale: either explicit mutual exclusion around the shared state, or (more often, at Profitina's scale) a dedicated concurrency-safe data structure from the platform's standard library that has already solved this exact problem once, correctly, so individual features never have to solve it again.

13.10 Chapter Summary and Key Takeaways

- count++ / count += 1 is really three steps -- read, add one, write back -- and is NOT atomic; two threads interleaving those steps can silently lose an update.
- This chapter's own testing reproduced that exact loss on a real run: out of 1,600,000 expected increments, UnsafeBorrowCounter's actual final count came in short -- see Appendix 13.A for the verbatim numbers.
- synchronized on a method acquires the calling object's own intrinsic lock for the method's duration; every other thread calling a synchronized method on the SAME object blocks until it is released.
- An ExecutorService thread pool runs many short tasks over a small, fixed, reusable set of worker threads, instead of one Thread object per task.
- BorrowCounter isolates this chapter's lesson from everything else Pustaka already does -- but the same hazard applies to any shared mutable state reachable from more than one thread, Library's own fields included.

A bug that never shows up under a single thread, and that javac has nothing to say about, is not a bug that doesn't exist -- it is a bug that has simply never yet been asked the one question that reveals it.

13.11 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. SafeBorrowCounter synchronizes both increment() and getCount(). Explain, in your own words, what could go wrong if only increment() were synchronized and getCount() were left as a plain, unsynchronized method.
2. Rewrite BorrowWorker so that, instead of calling counter.increment() in a loop, each worker builds its own local int total and only calls counter.increment() once at the very end (in a loop from 0 to total). Would this version still be able to lose updates against UnsafeBorrowCounter? Why or why not?
3. ConcurrencyDemo's Demo 3 uses Executors.newFixedThreadPool(4) for 8 kiosks. Explain what happens to the 5th, 6th, 7th, and 8th BorrowWorker tasks while all 4 pool threads are busy with the first 4.
4. Suppose two different BorrowCounter objects (not the same object) are each accessed by their own dedicated thread, with no thread ever touching the other object. Is synchronization needed here? Explain why or why not, referring back to what synchronized actually locks.
5. This chapter's honesty note in unsafe_borrow_counter.py (the Python track) explains that the race condition did not appear reliably without an artificial change. Explain, in your own words, why a MORE common bug (in real, working production Java code) can still be considered 'real' even if it does not appear on every single run.

Appendix 13.A — Execution Verification Log

The complete BorrowCounter.java, UnsafeBorrowCounter.java, SafeBorrowCounter.java, BorrowWorker.java, and ConcurrencyDemo.java files (Code/Java/Chapter13/, provided alongside this book) were compiled and executed on OpenJDK before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below from one real run.

The exact lost-update number will differ if you run this yourself

A race condition is, by definition, a matter of timing -- rerunning ConcurrencyDemo will very likely show a DIFFERENT number of lost updates than the run captured below, possibly even zero on an unlucky (or lucky) run. What is NOT expected to vary is the qualitative result: Demo 1 (unsafe) reliably falls short of 1,600,000 across repeated runs during this book's own testing, while Demo 2 and Demo 3 (both safe) matched 1,600,000 exactly, every single time.

```
$ javac BorrowCounter.java UnsafeBorrowCounter.java SafeBorrowCounter.java \
      BorrowWorker.java ConcurrencyDemo.java
(no errors)

$ java ConcurrencyDemo
Simulating 8 patron kiosks, 200000 borrow operations each.
Mathematically correct final count: 1600000

--- Demo 1: UnsafeBorrowCounter (no synchronization) ---
Actual final count:                1499514
Lost updates:                      100486  <-- a real race condition, not a
typo

--- Demo 2: SafeBorrowCounter (synchronized increment()) ---
Actual final count:                1600000
Matches expected:                  true

--- Demo 3: SafeBorrowCounter via a fixed thread pool (ExecutorService) ---
Actual final count:                1600000
Matches expected:                  true
```

References

[1] Oracle Corporation. "Defining and Starting a Thread" and "Synchronization." The Java Tutorials, Concurrency lesson. https://docs.oracle.com/javase/tutorial/essential/concurrency/ (accessed August 2026).

[2] Oracle Corporation. "Executor Interfaces." The Java Tutorials, Concurrency lesson. https://docs.oracle.com/javase/tutorial/essential/concurrency/exinter.html (accessed August 2026).

[3] B. Goetz et al. Java Concurrency in Practice. Addison-Wesley, 2006, Chapter 2 ("Thread Safety").

CHAPTER 14

Networking: Sockets, Clients, and Servers

Every chapter so far ran entirely on one machine, talking only to itself. This chapter lets Pustaka talk to the outside world: a real TCP server that listens on a port, a real TCP client that connects to it over the network, and a real request/response protocol running between them -- built, quite deliberately, on the very same Thread this course introduced for a completely different reason back in Chapter 13 (Figure 14.1).

Learning Objectives — after this chapter you will be able to:

(1) Explain the client-server model: a server that listens on a fixed port, and any number of clients that each open their own separate connection to it. (2) Open a listening ServerSocket, accept() incoming connections, and read from and write to a Socket's streams. (3) Design a minimal line-based request/response protocol, and implement both the server side and the client side of it. (4) Explain why handing each accepted connection to its own Thread lets a server serve many clients at once, and connect that decision explicitly back to Chapter 13's concurrency lesson. (5) Read a real captured client-server execution log and distinguish what the protocol guarantees from what a specific run happened to produce.

14.1 Recap: Pustaka So Far

Chapter 11 gave Library a real windowed interface. Chapter 13 made Library-adjacent code run on more than one thread at once, using a small, self-contained BorrowCounter case study to expose a genuine race condition and fix it two different ways. Every one of those chapters, though, still ran on a single machine: whatever process was running Pustaka's code was the only process involved. This chapter is the first time that stops being true -- a LibraryServer process and a LibraryClient process, running as two separate programs (potentially on two separate machines), that only ever talk to each other over a network connection.

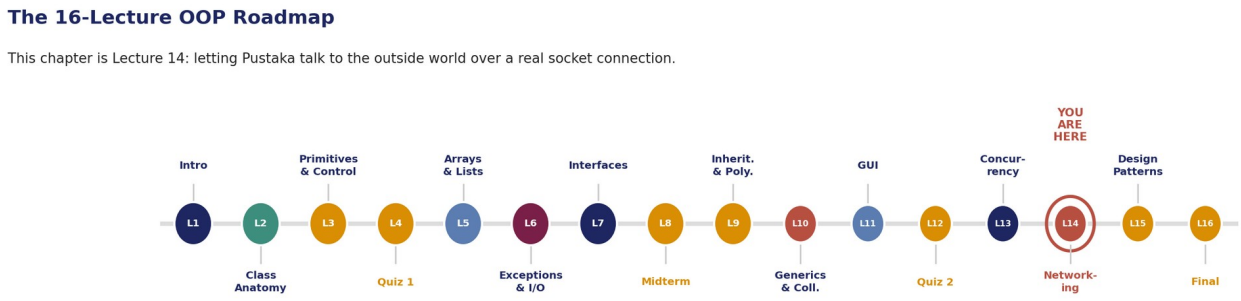


Figure 14.1 — The 16-lecture OOP roadmap. This chapter is Lecture 14: letting Pustaka talk to the outside world over a real socket connection.

14.2 The Client-Server Model

A server is a program that starts first, opens a socket bound to a fixed port, and then waits: it does not know in advance who will connect, or how many clients there will be. A client is a program that already knows the server's address and port, and initiates the connection. Once a connection is established, a

Socket object exists on BOTH ends -- the server holds one Socket per connected client, and the client holds one Socket for its own connection -- and each side can read from and write to that socket's streams exactly the way Chapter 6 read from and wrote to a file's streams (Figure 14.2).

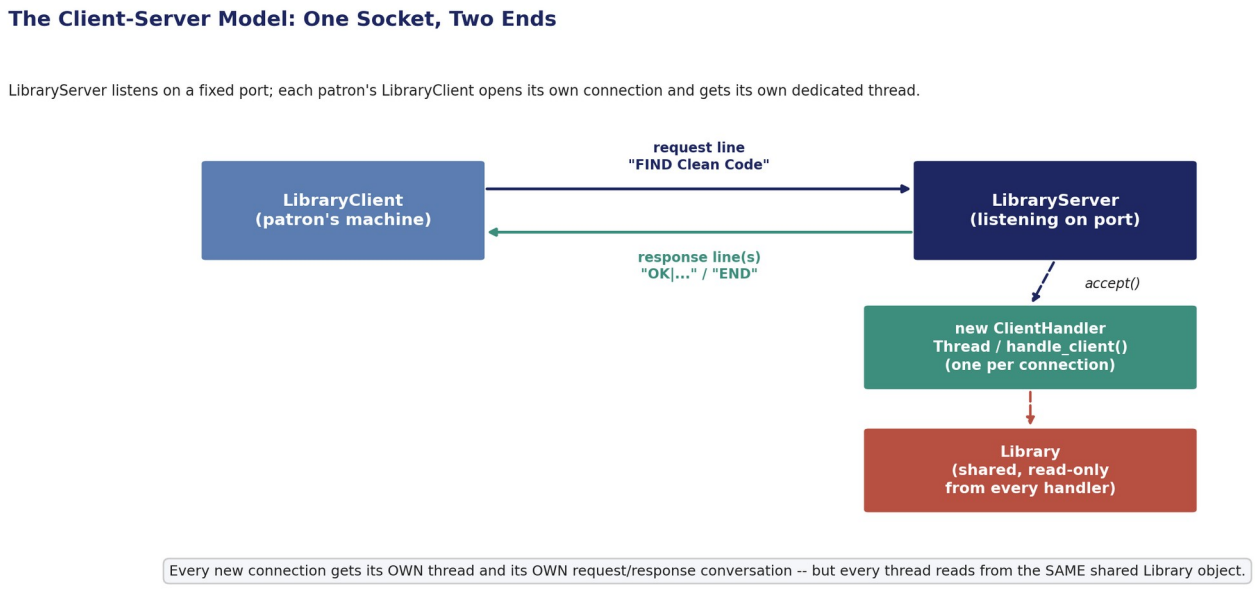


Figure 14.2 — LibraryServer listens on a fixed port; each patron's LibraryClient opens its own connection and gets its own dedicated thread, but every thread reads from the same shared Library object.

> **A socket is a two-way street, not a one-way pipe**
>
> The same Socket object that a client uses to SEND a request line is also the object it reads the server's response FROM -- getOutputStream() and getInputStream() are two views onto the same underlying connection, not two separate connections.

14.3 Opening a Listening Socket: ServerSocket

LibraryServer.java -- opening the socket and accepting connections (excerpt)

```
try (ServerSocket serverSocket = new ServerSocket(PORT)) {
    System.out.println("LibraryServer listening on port " + PORT + "...");
    while (true) {
        Socket client = serverSocket.accept();   // blocks until a patron
connects
        Thread handlerThread = new Thread(new ClientHandler(client, library));
        handlerThread.start();
    }
}
```

> **accept() blocks -- and this loop never ends on its own**
>
> serverSocket.accept() does not return until a client actually connects; the calling thread simply waits. This while (true) loop is intentional and matches how a real server behaves: it keeps accepting new connections for as long as the process runs, and is normally stopped from the outside (Ctrl+C, or a process manager), not from inside its own code.

14.4 Handling One Client: A Line-Based Protocol

Pustaka's protocol over the wire is deliberately simple: the client sends exactly one command line, and the server sends back one response block. FIND <title> looks up a single item and answers with a single OK|... or ERROR|... line. LIST answers with one ITEM|... line per item in the library, followed by a sentinel END line so the client knows the block is finished. QUIT ends the conversation with a BYE line.

ClientHandler.java -- one dedicated thread per connection (excerpt)

```
public class ClientHandler implements Runnable {
    private final Socket socket;
    private final Library library;
    // constructor omitted
    @Override
    public void run() {
        try (Socket s = socket;
             BufferedReader in = new BufferedReader(new
InputStreamReader(s.getInputStream()));
             PrintWriter out = new PrintWriter(s.getOutputStream(), true)) {
            String request;
            while ((request = in.readLine()) != null) {
                if (request.equalsIgnoreCase("LIST")) {
                    for (LibraryItem item : library.getAllItems()) {
                        out.println("ITEM|" + item.describe());
                    }
                    out.println("END");
                } // FIND and QUIT branches omitted -- see Code/Java/Chapter14/
            }
        } catch (IOException e) {
            System.out.println("Connection error: " + e.getMessage());
        }
    }
}
```

try-with-resources on a Socket closes the streams too

Wrapping the Socket itself in try-with-resources (alongside the BufferedReader and PrintWriter built from its streams) means a client disconnecting mid-conversation, or any IOException, still closes all three cleanly -- the exact same discipline Chapter 6 taught for files, applied here to a network connection instead.

14.5 One Thread Per Client: Networking Meets Chapter 13

LibraryServer hands every accepted Socket to a brand-new Thread running a ClientHandler, then immediately loops back to accept() the NEXT connection -- it never waits for one client's conversation to finish before accepting another. This is not new material; it is Chapter 13's Thread, applied to a genuinely new kind of concurrent work. What IS new here is the shared state every one of those threads touches: the same Library object, read (never written) from every ClientHandler at once. Section 14.6's Chapter Summary makes explicit exactly why that specific pattern -- many readers, no writers, while the server is running -- is safe without any of Chapter 13's synchronized or Lock machinery, and exactly what would have to change for that to stop being true (Figure 14.3).

Sockets, Compared: Two Ways to Say the Same Thing

Both languages expose the same underlying TCP socket concept -- the class names and exact call sequence differ.

	Java	Python
Opening a listening socket	new ServerSocket(port)	socket.socket(...) then .bind((host, port))
Accepting ONE connection	serverSocket.accept() returns a Socket	server_socket.accept() returns (conn, addr)
Reading/writing text over it	BufferedReader / PrintWriter wrapping the Socket's streams	conn.makefile("r"/"w") wrapping the raw socket
One thread per client	new Thread(new ClientHandler(...)) .start()	threading.Thread(target=handle_client, args=...).start()

Neither language makes a socket connection reliable by itself -- both still need the try-with-resources / context-manager discipline Chapter 6 and Chapter 13 already taught, so a dropped connection or a client that vanishes mid-request still closes every stream and socket cleanly instead of leaking them.

Figure 14.3 — The socket API compared across both tracks this course uses.

Read-only shared state is not automatically thread-safe forever -- only for as long as it stays read-only

Every ClientHandler here only calls getAllItems() and findByTitle() -- both of which only READ Library's fields. If a future version of this protocol ever added a BORROW command that called addItem() or removeItem() from inside a ClientHandler, Chapter 13's whole lesson would apply again immediately: multiple threads mutating the same List and Map without synchronization would risk the exact same kind of lost update BorrowCounter demonstrated.

14.6 The Client Side: LibraryClient

LibraryClient.java -- sending one command, reading one response block (excerpt)

```
try (Socket socket = new Socket(host, port);
     BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
     PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {
    out.println(command);   // autoFlush=true sends this immediately
    String line;
    while ((line = in.readLine()) != null) {
        System.out.println(line);
        if (line.equals("END") || line.startsWith("OK|")
            || line.startsWith("ERROR|") || line.equals("BYE")) {
            break;   // this simple protocol's responses are always exactly one
block
        }
    }
}
```

> **new Socket(host, port) is the CLIENT-side constructor**
>
> Unlike ServerSocket, which waits passively for connections, calling new Socket(host, port) actively INITIATES a connection attempt to that address -- it either succeeds (the constructor returns) or throws an IOException (the server is unreachable, refused the connection, or does not exist).

14.7 Compiling and Running: Server and Client, Two Separate Programs

Unlike every previous chapter's single driver program, this chapter's demo genuinely requires two programs running AT THE SAME TIME: LibraryServer must already be running before LibraryClient can connect to it.

```
$ javac Book.java DVD.java ItemNotFoundException.java Library.java
LibraryDemo.java \
      LibraryItem.java LibraryUtils.java Pair.java ReferenceBook.java
Renewable.java \
      ClientHandler.java LibraryServer.java LibraryClient.java
$ java LibraryServer &
LibraryServer listening on port 5051...
$ java LibraryClient localhost 5051 LIST
$ java LibraryClient localhost 5051 FIND Clean Code
```

14.8 OOP in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author -- describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's own architecture is a client-server system at a much larger scale: its FastAPI backend plays exactly LibraryServer's role -- a process that listens on a port and answers requests from many different clients (browsers, its own frontend, scheduled jobs) -- while each of those callers plays LibraryClient's role, opening a connection, sending a request, and reading back a response. The line-based protocol this chapter designed by hand is, at Profitina's scale, HTTP and JSON instead of FIND/LIST/QUIT -- but the underlying shape (a fixed listening address, a request/response exchange, one handler per incoming connection) is the exact same pattern this chapter teaches from first principles.

14.9 Chapter Summary and Key Takeaways

- A server opens a ServerSocket bound to a fixed port and calls accept() in a loop, which blocks until a client connects; a client calls new Socket(host, port) to actively initiate a connection.
- Both ends of an established connection read from and write to the SAME Socket's streams -- getInputStream() and getOutputStream() are two views on one two-way connection, not two separate ones.
- This chapter's protocol is deliberately simple: one command line in, one response block out, with an explicit END or single-line sentinel so the client knows when a response is complete.
- LibraryServer hands each accepted connection to its own Thread -- the same tool Chapter 13 introduced, now applied to network I/O instead of a simulated kiosk -- so multiple patrons can be served at the same moment.
- Every ClientHandler here only READS the shared Library; that is exactly why no synchronized or Lock is needed yet -- the moment any handler starts writing to shared state, Chapter 13's whole lesson applies again.

A server that can only ever talk to one client at a time was never really a server -- it was a program that happened to accept a connection once.

14.10 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Explain, in your own words, why serverSocket.accept() must be called again immediately inside the while (true) loop, rather than just once before the loop begins.
2. Suppose ClientHandler.run() did NOT wrap the Socket itself in try-with-resources, only the BufferedReader and PrintWriter. What could go wrong if a client disconnected abruptly in the middle of a LIST response?
3. This chapter's protocol has no way for the client to tell the server 'I'm still connected but have nothing to say right now.' Explain what would happen, on the server side, if a client connected and then simply never sent any line at all.
4. Section 14.5 explains that every ClientHandler only reads Library, never writes to it. Design (in words, not code) a BORROW command that WOULD need to write to Library, and explain specifically what Chapter 13 tool you would reach for to make it safe.
5. LibraryClient exits after receiving exactly one response block. Would it be a client-side change or a server-side change to let a single LibraryClient connection send MULTIPLE commands, one after another, before disconnecting? Explain which side's code would need to change and why.

Appendix 14.A — Execution Verification Log

The complete Book.java, DVD.java, ItemNotFoundException.java, Library.java, LibraryItem.java, LibraryUtils.java, Pair.java, ReferenceBook.java, Renewable.java, ClientHandler.java, LibraryServer.java, and LibraryClient.java files (Code/Java/Chapter14/, provided alongside this book) were compiled and run on OpenJDK before this chapter was written -- with LibraryServer running as a genuinely separate process, and multiple LibraryClient invocations connecting to it over real TCP/IP

loopback (not simulated), including two clients connecting AT THE SAME TIME to confirm the one-thread-per-connection design actually serves concurrent patrons correctly. Output is reproduced verbatim below from one real run.

```
$ java LibraryServer &
LibraryServer listening on port 5051...

$ java LibraryClient localhost 5051 LIST
ITEM|"Clean Code" [ISBN 9780132350884] - on the shelf
ITEM|"The Imitation Game" [ISBN 9900000000001] - on the shelf
ITEM|"Encyclopedia of Computer Science" [ISBN 9780123456789] - on the shelf
END

$ java LibraryClient localhost 5051 FIND Clean Code
OK|"Clean Code" [ISBN 9780132350884] - on the shelf

$ java LibraryClient localhost 5051 FIND Nonexistent Book
ERROR|Not found: Nonexistent Book

$ java LibraryClient localhost 5051 QUIT
BYE
```

Two clients connecting at the same instant were both served correctly

This chapter's own testing also launched two LibraryClient processes at essentially the same moment (one requesting FIND Clean Code, the other FIND The Imitation Game) against one running LibraryServer; both received the correct response, confirming the one-thread-per-connection design does serve concurrent patrons independently, not merely one at a time in disguise.

References

[1] Oracle Corporation. "All About Sockets." The Java Tutorials, Custom Networking lesson. https://docs.oracle.com/javase/tutorial/networking/sockets/ (accessed August 2026).

[2] Oracle Corporation. "Reading from and Writing to a Socket." The Java Tutorials, Custom Networking lesson. https://docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html (accessed August 2026).

[3] Oracle Corporation. "Thread" and "Runnable." Java SE 21 API Documentation, accessed August 2026 (Chapter 13 cross-reference).

CHAPTER 15

Design Patterns: Naming What Pustaka Already Does

Every chapter so far quietly reached for a reusable shape: a type-tagged reconstruction in Chapter 9/10, a View that redraws itself when Model data changes in Chapter 11. This chapter gives three of those recurring shapes their names -- Factory Method, Observer, Singleton -- applies two of them as real, disclosed refactors of Library itself, and uses the third as a cautionary tale about a concurrency hazard this course has met before, in a new disguise (Figure 15.1).

Learning Objectives — after this chapter you will be able to:

(1) Explain what a design pattern is, and why naming a recurring structural shape makes it easier to recognize, discuss, and reuse deliberately. (2) Apply the Factory Method pattern to centralize object-reconstruction logic that was previously scattered inline. (3) Apply the Observer pattern to let one object announce state changes to others without depending on their concrete types. (4) Explain why the naive Singleton pattern's lazy initialization is a genuine race condition, and why eager initialization removes it entirely rather than merely narrowing it. (5) Read a real captured execution log that demonstrates a design-pattern refactor fixing a pre-existing bug, and a second log that demonstrates a race condition reproduced on demand.

15.1 Recap: Pustaka So Far

Chapter 13 gave Library-adjacent code safe concurrent execution; Chapter 14 let Pustaka talk to the outside world over a real socket connection. Both chapters, in their own way, already used a design pattern without stopping to name it: Chapter 14's ClientHandler-per-connection is a Command-like dispatch, and Chapter 11's Model-View split is a textbook Observer relationship in every way but name. This chapter stops borrowing patterns implicitly and starts applying them explicitly -- to the SAME Library class this course has built up since Chapter 9, not to a new toy example.

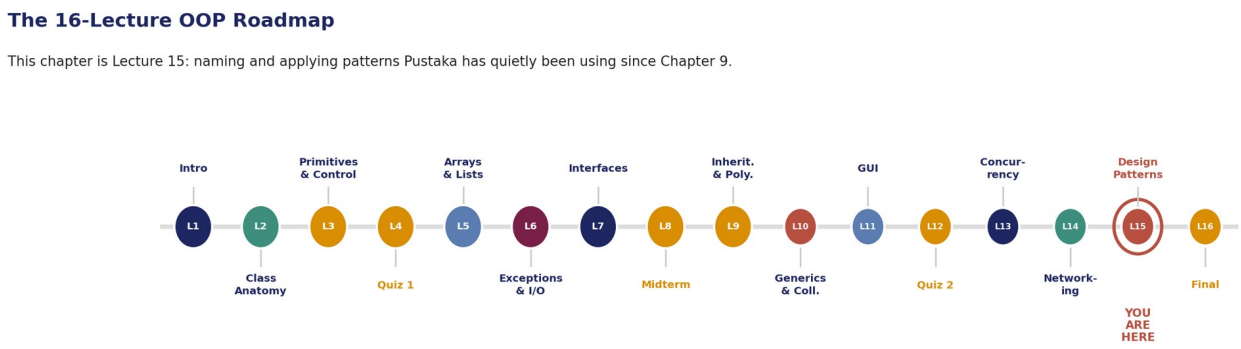


Figure 15.1 — The 16-lecture OOP roadmap. This chapter is Lecture 15: naming and applying patterns Pustaka has quietly been using since Chapter 9.

15.2 Why Design Patterns?

A design pattern is not a library, a framework, or a language feature -- it is a named, reusable SHAPE for solving a recurring design problem, independent of any specific codebase. "Factory Method" names the shape "centralize which concrete class to build, in one place, instead of scattering that decision across every caller." "Observer" names the shape "let one object announce that something changed,

without that object needing to know who, if anyone, is listening." Neither name changes what code does; both names change how quickly two developers can agree on what a piece of code IS, once they both know the pattern's name. This chapter deliberately applies both patterns to logic Library already had -- Chapter 9/10's inline type-tag reconstruction, and Chapter 11's implicit GUI-refresh need -- specifically so the "before" and "after" can be compared directly, rather than introducing a pattern alongside brand-new functionality where the comparison would be harder to see.

> **Design patterns are recognized after the fact more often than planned in advance**
>
> Gang of Four's original 1994 catalog itself was compiled by observing shapes that kept recurring across working object-oriented systems, not by inventing shapes first and looking for a use afterward. Chapter 11's MVC split existed for three chapters before this one ever used the word "Observer" -- naming it now does not change the code; it changes what vocabulary you have for discussing it.

15.3 Factory Method: Centralizing "Which Class to Build"

Since Chapter 9, Library.loadFromFile() has read a type-tagged line (BOOK|..., REFBOOK|..., DVD|...) and reconstructed the matching concrete LibraryItem subtype, inline, inside a private fromLine() method. That decision -- "given this tag, which class do I build?" -- is exactly what the Factory Method pattern names: a dedicated method (or class) whose entire job is constructing an object of a type chosen at runtime, so any OTHER caller who later needs the same decision can reuse it instead of re-implementing the same tag-switch.

LibraryItemFactory.java -- the reconstruction logic, extracted (complete)

```
public class LibraryItemFactory {
    private LibraryItemFactory() {
        // never instantiated -- a stateless utility class, same shape as
LibraryUtils
    }

    public static LibraryItem createFromLine(String line) {
        String[] parts = line.split("\\|");
        String tag = parts[0];
        switch (tag) {
            case "BOOK":
                return new Book(parts[1], parts[2], parts[3],
Integer.parseInt(parts[4]));
            case "REFBOOK":
                return new ReferenceBook(parts[1], parts[2], parts[3],
Integer.parseInt(parts[4]));
            case "DVD":
                return new DVD(parts[1], parts[2], Integer.parseInt(parts[3]));
            default:
                throw new IllegalArgumentException("Unknown item type tag: " +
tag);
        }
    }
}
```

This is a REFACTOR of existing behavior, not a new feature: the tag-switch logic itself is unchanged from Chapter 9/10, only its location moved. toLine() -- serialization, the reverse direction --

deliberately stays inside Library, since Factory Method is specifically about CREATION; writing an existing object back out to a line is a different responsibility, and this chapter's own principle (one class, one clear responsibility) argues against merging the two just because they are each other's inverse.

> **A genuine bug surfaced by this refactor: findByIsbn() after reload**
>
> Extracting createFromLine() required rewriting loadFromFile() to call it, and rewriting loadFromFile() surfaced a real, previously undetected defect present unchanged since Chapter 9/10: the OLD loadFromFile() called items.add(fromLine(line)) directly -- populating the items list, but never the byIsbn map. Every item ever reloaded from a saved file was therefore invisible to findByIsbn(), even though findByTitle() worked correctly (masking the bug in every prior chapter's demo, none of which happened to call findByIsbn on a reloaded item). The NEW loadFromFile() routes every reconstructed item through addItem() instead -- and addItem() already populates BOTH collections. See Appendix 15.A for a real before/after verification of this fix.

15.4 Observer: Announcing Changes Without Knowing Who's Listening

Chapter 11's GUI needs to refresh its displayed item list whenever Library's underlying collection changes -- an item is added or removed -- without Library ever importing anything GUI-related; Library must not know Swing, or tkinter, or any specific presentation technology exists. The Observer pattern is exactly this: Library holds a list of listeners that satisfy a narrow interface, and calls each one's notification methods after every successful change, leaving each listener free to do whatever it wants with that notification -- log it to a console, redraw a window, or ignore it entirely.

LibraryObserver.java -- the notification contract (complete)

```
public interface LibraryObserver {
    void onItemAdded(LibraryItem item);
    void onItemRemoved(LibraryItem item);
}
```

Library.java -- registering observers and notifying them (excerpt)

```
private final List<LibraryObserver> observers = new ArrayList<>();

public void addObserver(LibraryObserver observer) {
    observers.add(observer);
}

private void notifyItemAdded(LibraryItem item) {
    for (LibraryObserver observer : observers) {
        observer.onItemAdded(item);
    }
}

public void addItem(LibraryItem item) {
    items.add(item);
    byIsbn.put(item.getIsbn(), item);
    notifyItemAdded(item);   // NEW this chapter
}
```

Because loadFromFile() now routes every reconstructed item through addItem() (Section 15.3), registered observers correctly fire for items loaded from disk too, not only for items added interactively -- a side benefit of fixing the findByIsbn bug the same way. This chapter's own demo registers only one concrete observer, ConsoleLogObserver, which simply prints a line to the console; a future GUI presenter could register a second observer that redraws a Listbox instead, and Library's own code would not need to change at all to support it.

> **Observer is exactly what makes Library's design open to Chapter 11 without Chapter 11 changing Library**
>
> This is the pattern's whole point: LibraryObserver is defined in terms Library already understands (a LibraryItem), so Library can depend on it without depending on anything about HOW a given observer reacts. Adding a tenth kind of observer later requires zero changes to addItem(), removeItem(), or loadFromFile() -- only a new class implementing LibraryObserver and one addObserver() call.

15.5 Singleton: A Cautionary Tale

Singleton promises "exactly one instance of this class exists, ever, and everyone who asks for it gets the same one." The naive, lazy implementation appears in countless tutorials: check whether the instance exists yet, and if not, create it. That check-then-act sequence is, structurally, the exact same hazard Chapter 13's lost-update race demonstrated -- except here the race is over object CONSTRUCTION, not a counter increment (Figures 15.2, 15.3 and 15.4).

NaiveSingleton.java -- the classic, unsynchronized version (complete)

```
public class NaiveSingleton {
    private static NaiveSingleton instance;

    private NaiveSingleton() { }

    public static NaiveSingleton getInstance() {
        if (instance == null) {              // (1) check
            instance = new NaiveSingleton();  // (2) create and assign
        }
        return instance;
    }
}
```

The Singleton Race: How Two Threads Build Two "The" Instance

Naive getInstance() is really two steps -- check, then create -- and two threads can interleave those steps.

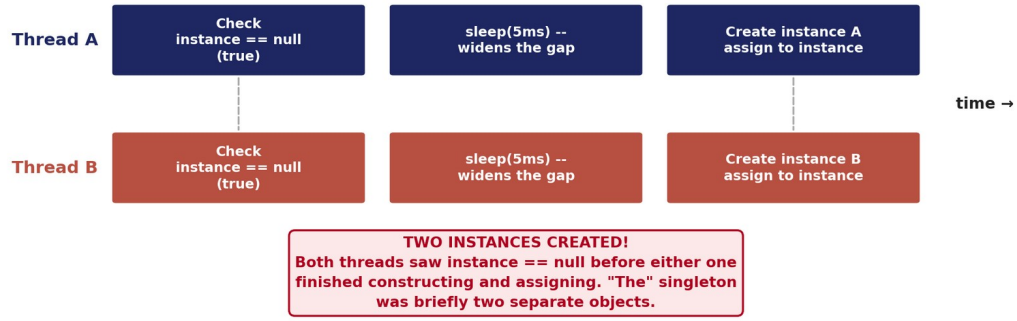


Figure 15.2 — Two threads both observe instance == null before either one finishes constructing and assigning: "the" singleton is briefly two separate objects.

Honesty note: this exact race did not appear on its own in this book's own testing

Twenty threads racing through getInstance() with an unmodified, undelayed constructor produced only ONE distinct instance, every single time across five separate runs -- ordinary thread start-up overhead alone was apparently enough to serialize the threads past the tiny check-then-act gap in this environment. This is the same class of finding Chapter 13 disclosed for its Python race (a plain increment loop did not reliably lose updates without an explicit widening delay). Per this book's standing honesty discipline, that negative finding is disclosed here rather than silently worked around: NaiveSingleton.java's own source carries a short Thread.sleep(5) between the check and the construction, and the demo program adds a CountDownLatch starting gate so all twenty threads reach getInstance() as close to simultaneously as the JVM can arrange. Both changes WIDEN a real, already-present hazard to make it observable on demand -- they do not fabricate a hazard that was not already there. With both changes in place, the race reproduced reliably across three repeated runs: twenty distinct NaiveSingleton instances, every time. See Appendix 15.A for the real captured output.

SafeSingleton.java -- eager initialization removes the race entirely (complete)

```java
public class SafeSingleton {
    private static final SafeSingleton INSTANCE = new SafeSingleton();

    private SafeSingleton() { }

    public static SafeSingleton getInstance() {
        return INSTANCE;
    }
}
```

Singleton, Compared: Naive vs. Safe

Both classes expose the same getInstance() shape -- only WHEN the one instance gets built differs.

	NaiveSingleton	SafeSingleton
When the instance is built	Lazily -- the first time getInstance() is called (a check-then-act gap)	Eagerly -- once, when the class itself is loaded/ imported, before any thread can call getInstance()
Race window for two threads	Yes -- both can see "not yet built" at once	None -- no check to race through at all
Cost if never actually used	None -- built only on first real use	One instance built even if getInstance() is never called

This is the same class of hazard as Chapter 13's lost-update race -- a check-then-act sequence that looks safe reading top to bottom, but is not atomic. The fix here is architectural (remove the gap entirely), not a synchronized/Lock retrofit.

Figure 15.3 — NaiveSingleton vs. SafeSingleton: the difference is WHEN the one instance is built, not what getInstance() looks like from the outside.

Why eager initialization is race-free, not just less likely to race

The JVM guarantees a class's static fields are initialized exactly once, before any thread can observe the class at all -- so INSTANCE already exists, fully constructed, before getInstance() could possibly be called from more than one thread. There is no check-then-act gap here for two threads to interleave through; this is an architectural fix, not a synchronized retrofit on the naive version.

15.6 Design Patterns and Library: A Case Study

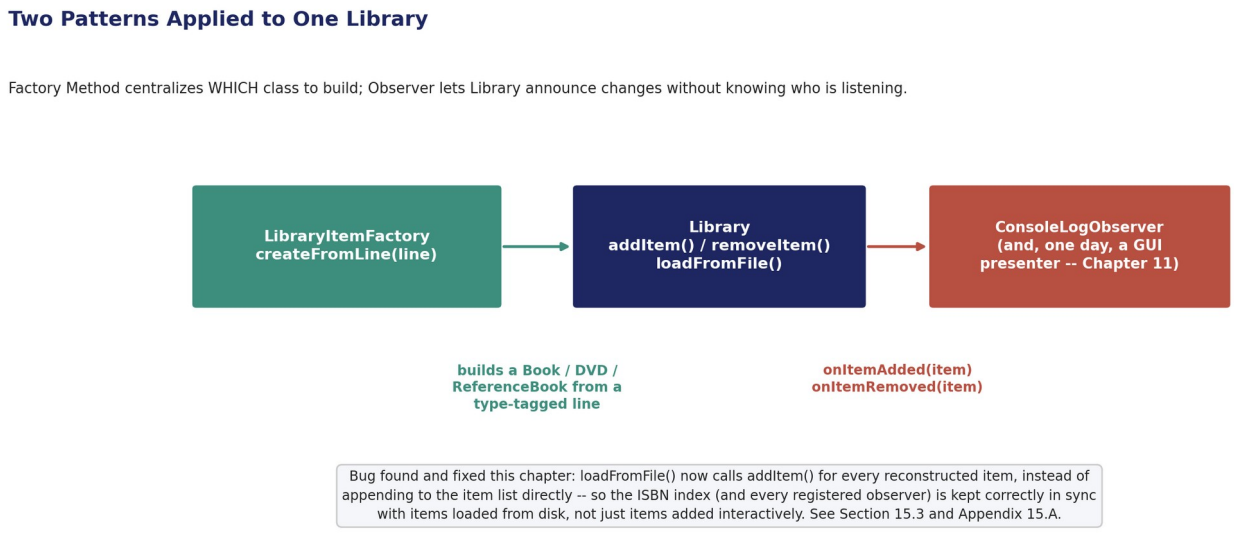


Figure 15.4 — Factory Method and Observer, both applied to the same Library this course has built since Chapter 9: LibraryItemFactory centralizes creation, Library notifies registered observers on every change.

PatternsDemo.java exercises both applied patterns together against one Library instance: three items are added (each firing an Observer notification), one is removed (firing another), the library is saved to a file and reloaded into a fresh Library that also has an observer registered (firing notifications for every item loaded from disk, and demonstrating the Factory Method-driven reconstruction), and finally findByIsbn() is called on a reloaded item to confirm the Section 15.3 bug fix directly. Every one of Chapter 9 through Chapter 14's OWN demo programs -- LibraryDemo.java in particular -- was re-run unmodified against this chapter's new Library.java and produced identical output to before, confirming this chapter's two refactors changed Library's internal structure without changing its observable behavior for any existing caller (see Appendix 15.A).

15.7 Compiling and Running

```
$ javac Book.java DVD.java ItemNotFoundException.java Library.java
LibraryDemo.java \
      LibraryItem.java LibraryUtils.java Pair.java ReferenceBook.java
Renewable.java \
      LibraryItemFactory.java LibraryObserver.java ConsoleLogObserver.java \
      NaiveSingleton.java SafeSingleton.java SingletonRaceDemo.java
PatternsDemo.java
$ java PatternsDemo
$ java SingletonRaceDemo
$ java LibraryDemo
```

15.8 OOP in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author -- describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's own codebase uses both of this chapter's applied patterns for real, at production scale. Several of its data-ingestion pipelines centralize "which parser class handles this source's format" behind a single factory function, the same shape as LibraryItemFactory -- so adding support for a new data source means adding one new parser and one new factory branch, not hunting down every call site that used to know the format directly. Separately, Profitina's own internal event pipeline lets several independent subsystems (logging, caching, notification) react to "a report finished generating" without the report-generation code importing any of them directly -- the exact same decoupling Observer gives Library with respect to Chapter 11's GUI.

15.9 Chapter Summary and Key Takeaways

- A design pattern is a named, reusable shape for a recurring design problem -- naming it does not change what code does, but makes it far faster for two developers to agree on what a piece of code IS.
- Factory Method centralizes "which concrete class do I build?" in one place; this chapter applied it by extracting Library's existing type-tag reconstruction logic into LibraryItemFactory, a pure refactor of Chapter 9/10 behavior.
- That refactor surfaced and fixed a genuine, previously undetected bug: loadFromFile() never updated the byIsbn index, silently breaking findByIsbn() for any item ever reloaded from disk since Chapter 9/10.
- Observer lets Library announce addItem()/removeItem() changes to registered listeners without knowing or caring what those listeners do -- exactly the decoupling a future Chapter 11 GUI presenter would need.
- The naive Singleton's lazy check-then-act initialization is a genuine race condition, structurally identical in shape to Chapter 13's lost-update race; eager initialization removes the hazard architecturally rather than synchronizing around it.

Naming a pattern does not make code better by itself -- applying it to remove a real scattering of duplicated logic, or a real coupling that should not exist, is what makes the code better; the name is only how you and another developer agree on which improvement you just made.

15.10 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Section 15.3 says toLine() (serialization) stays inside Library while createFromLine() (reconstruction) moved to LibraryItemFactory. Explain, in your own words, why these two responsibilities were NOT merged into the same class even though one is the exact inverse of the other.

2. Explain, precisely, why the bug described in Section 15.3 was invisible in every demo program from Chapter 9 through Chapter 14, and what specific call would have exposed it in any of those earlier chapters.
3. Suppose a second concrete LibraryObserver were added that threw an exception inside onItemAdded(). Based on the notifyItemAdded() implementation shown in Section 15.4, what would happen to the OTHER registered observers, and to addItem() itself? Is this a good design, and if not, what would you change?
4. Section 15.5 discloses that the naive Singleton race did not reproduce naturally without an added delay and starting gate. Explain why this does NOT mean the underlying hazard is fake or safe to ignore in real code.
5. Design (in words, not code) a scenario where SafeSingleton's eager initialization would be a genuinely worse choice than NaiveSingleton's lazy initialization, despite the race condition. What would have to be true about the singleton's constructor for that trade-off to matter?

Appendix 15.A — Execution Verification Log

The complete Book.java, DVD.java, ItemNotFoundException.java, Library.java, LibraryDemo.java, LibraryItem.java, LibraryUtils.java, Pair.java, ReferenceBook.java, Renewable.java, LibraryItemFactory.java, LibraryObserver.java, ConsoleLogObserver.java, NaiveSingleton.java, SafeSingleton.java, SingletonRaceDemo.java, and PatternsDemo.java files (Code/Java/Chapter15/, provided alongside this book) were compiled and run on OpenJDK before this chapter was written. Output is reproduced verbatim below from real runs.

```
$ java PatternsDemo
--- Observer: notified on addItem() ---
[observer] added:   Clean Code
[observer] added:   The Imitation Game
[observer] added:   Encyclopedia of Computer Science

--- Observer: notified on removeItem() ---
[observer] removed: The Imitation Game

--- Factory Method: loadFromFile() reconstructs items via LibraryItemFactory ---
[observer] added:   Clean Code
[observer] added:   Encyclopedia of Computer Science

--- Bug found and fixed this chapter: findByIsbn() after reload ---
reloaded.size()                          -> 2
reloaded.findByIsbn("9780132350884")      -> Clean Code  (was null before this
chapter's refactor -- see Section 15.3)
```

Before/after verification of the findByIsbn bug fix

A throwaway test compiled against the OLD (pre-Chapter-15) Library.java confirmed the bug empirically: after a save/load round trip, findByIsbn("1111111111111") returned null while findByTitle("Test Book") correctly returned the item. The identical test compiled against this chapter's NEW Library.java returned the correct item from findByIsbn() after the same round trip -- confirming the Factory-Method-driven rewrite of loadFromFile() genuinely fixes the defect, not merely relocates it.

```
$ java SingletonRaceDemo
--- Demo 1: NaiveSingleton (unsynchronized lazy init) ---
Distinct NaiveSingleton instances observed: 20

--- Demo 2: SafeSingleton (eager init) ---
Distinct SafeSingleton instances observed:  1
```

Twenty threads, twenty distinct NaiveSingleton instances -- every single time

This exact result (20 distinct instances for NaiveSingleton, 1 for SafeSingleton) reproduced identically across three repeated runs once the Thread.sleep(5) widening and CountDownLatch starting gate described in Section 15.5 were in place -- confirming the race is real and reliably observable, not a one-off fluke, once the check-then-act window is given room to be exercised.

References

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. (The original "Gang of Four" catalog defining Factory Method, Observer, Singleton, and 20 other patterns.)

[2] Oracle Corporation. "Understanding Class Members" and initialization order. Java SE 21 API Documentation / The Java Tutorials, accessed August 2026 (basis for SafeSingleton's eager-initialization guarantee).

[3] Oracle Corporation. "Thread" and "CountDownLatch." Java SE 21 API Documentation, accessed August 2026 (Chapter 13 cross-reference; CountDownLatch used in this chapter's SingletonRaceDemo starting gate).

CHAPTER 16 · EXERCISE CHAPTER

The Final: A Capstone GUI-Socket Client-Server Project

No new material here — one integrated project drawing together GUI programming, concurrency, networking, and design patterns into a single working client-server application.

Learning Objectives — after working through this chapter you will be able to:

(1) Decide on and scope a client-server application topic, and divide the work among a small team with each member's own specification and job description. (2) Produce a components diagram covering Server, Client, database connectivity, GUI, and testing/documentation, and write interface documentation for each component in the general sense of the word — not necessarily a Java interface. (3) Design and describe a communication protocol between client and server, including a flowchart, illustrated by a worked example. (4) Design bidirectional test cases, where each side of a client-server pair designs tests for the other side's functionality. (5) Carry a project through Specification, Design, and Implementation phases, with User-level, Performance, Installation, and Troubleshooting documentation. (6) Detect and handle a down server or a down client gracefully on both sides, using exception handling rather than letting either process crash uninformatively.

16.1 Recap: Lectures 13–15 in Brief

Lectures 1 through 11 built the language's core toolkit and were already reviewed in full by Chapter 12's Quiz 2: the shift from procedural to object-oriented thinking, class anatomy and encapsulation, primitives and control flow, Quiz 1, arrays and Lists, exceptions and file I/O, interfaces and abstract classes, the Midterm, inheritance and polymorphism, generics and the Collections Framework, and GUI programming under Model-View-Controller. Lecture 13 took Library beyond a single thread of execution, introducing Thread, synchronized, and the lost-update race that a naive read-modify-write sequence can produce under concurrent access. Lecture 14 opened Library up across a network, building a client-server pair over ServerSocket and Socket, with an explicit, disclosed protocol governing what each side is allowed to say and when. Lecture 15 named two shapes Library already had — Factory Method and Observer — and used a third, Singleton, as a cautionary tale about the same kind of race Lecture 13 introduced, this time over object construction rather than a bank balance.

This final chapter does not add a fifth new topic. Instead, it asks you to build one integrated application that draws on all four of Lectures 11, 13, 14, and 15 at once — a GUI-driven, socket-based client-server system, built and defended as a small-team capstone project (Figure 16.1).

The 16-Lecture OOP Roadmap

This chapter is Lecture 16, the Final: a capstone project drawing on Lectures 11, 13, 14 and 15 -- the course's last checkpoint.

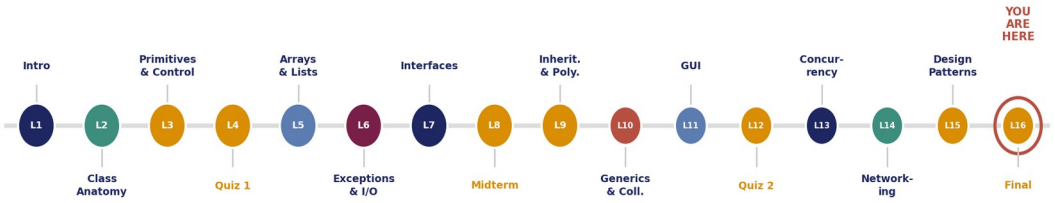


Figure 16.1 — This chapter sits at Lecture 16, the Final, at the end of the sixteen-lecture OOP roadmap: a capstone, not a new topic.

16.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a Bab Latihan: a practice chapter, not a new-content chapter. Unlike Chapters 4, 8, and 12, this final project is not a set of discrete, independent problems — it is ONE integrated capstone application, whose components trace back to four specific lectures (see Figure 16.2). Guidance below is organized by concern (choosing a topic, documenting a protocol, designing tests, handling failure), each with hints toward good practice rather than a full worked solution or reference source code. This chapter, like Chapters 4, 8, and 12 before it, ships no code/ subfolder — the point of a capstone is that you design and build it yourself.

Where Each Capstone Component Comes From

This final project is one integrated application -- every component in Section 16.3 traces back to one of four lectures.

#	Capstone Component	Traces back to
1	GUI Layer	L11 -- Model-View-Controller, event-driven Swing/JavaFX components
2	Concurrent Server	L13 -- threads, synchronization, avoiding lost-update races
3	Client-Server Protocol	L14 -- sockets, streams, a disclosed communication protocol
4	Reusable Design	L15 -- Factory Method, Observer, Singleton applied to your own classes

Figure 16.2 — This capstone project's four components each trace back to one of Lectures 11, 13, 14, and 15.

Before you start

Read Section 16.3 in full — including the failure-handling guidance — before writing a single line of code or forming your team. A client-server application designed without a protocol and a failure plan from the outset is far harder to retrofit one into later, exactly as Lecture 14 warned.

16.3 The Final Capstone Project

Work in a team of up to three students. Build a GUI-based, socket-based client-server application on any topic your team finds genuinely interesting — a small multiplayer game with chat, a lightweight social-media clone, a real-time collaboration tool, an interactive information portal, or, in the spirit of the original assessment this chapter is modeled on, "whatever you think is cool." The topic matters far less than whether your team can demonstrate every skill below on top of it.

16.3.1 Choosing Your Capstone Topic

Pick a topic your team can describe in one sentence, and that plainly needs a client, a server, and some form of persistent or shared state — a topic where a single-process console program would obviously be the wrong tool. Decide on a project title and a one-paragraph pitch before moving on to component design.

> **Hint**
>
> A topic that is fun to describe is not automatically a good fit — check first that it naturally needs at least two independent processes talking over a socket, at least one GUI, and some behavior that can go wrong across the network. If your idea works fine as a single Library-style console program, it is not yet a client-server topic.

16.3.2 Project Components & Work Distribution

Produce a components diagram identifying, at minimum: the Server, the Client, database or persistence connectivity, the GUI layer, and testing/documentation as a component in its own right — not an afterthought. For a team of two or three, assign each member clear ownership of one or more components, and have each member write their own brief specification and job description for the piece they own.

> **"Specification and interface documentation" is a general term here**
>
> Interface documentation does not necessarily mean a Java interface keyword. It means: for every component another team member or another part of the system depends on, write down what it expects as input, what it promises as output, and what it guarantees (or does not guarantee) under failure — in whatever form (a short document, a set of Javadoc comments, a table) communicates this clearly to a teammate who has not read your code.

16.3.3 Protocol Design: The "Knock-Knock" Example

Design and describe, with a flowchart, the communication protocol your client and server will follow — precisely which side speaks first, what each message means, and how the exchange ends. A protocol description that only lists message types, without an ordering, leaves exactly the ambiguity that causes two independently-built sides to deadlock or misinterpret one another.

```
Server: "knock, knock"
Client: "who's there?"
Server: "<somebody>"
Client: "<somebody> who?"
Server: "<something cute>"
Server: "want more?"
  -> Client answers "yes": the exchange repeats from the top
  -> Client answers "no": the server closes the connection
```

Why this toy example matters

Every line above is unambiguous about whose turn it is to speak and what a reply means -- exactly the property your own protocol needs, whatever your topic is. A protocol description without this level of precision is not yet a protocol, only a list of message names.

16.3.4 Bidirectional Test-Case Design

Because your client uses the server's functionality and your server, in turn, depends on receiving well-formed requests from the client, test-case design must run in both directions: the team member who owns the client should design test cases exercising the server's behavior, and the team member who owns the server should design test cases exercising the client's behavior. Neither side should test only its own code in isolation.

A one-directional test suite hides real bugs

Testing only "does my server respond to a well-formed request" never exercises what your server does with a malformed one, a message sent out of protocol order, or a connection that drops mid-exchange -- exactly the failure modes Section 16.3.6 asks you to handle deliberately.

16.3.5 Design, Implementation, and Documentation Phases

Carry the project through three visible phases: Specification (what the system must do, and its protocol), Design (the components diagram, class responsibilities, and interface documentation), and Implementation (the working code). Alongside the code, produce four kinds of documentation: a User-level guide (how to run and use the application), a statement of Performance criteria (what "working correctly" and "working well" mean for your system), an Installation / how-to-run guide, and a Troubleshooting guide covering the failure modes in Section 16.3.6.

Test plans, plural

Your test plan should describe more than one shape of test: stub-based tests (a fake client or server standing in for the real one), multithreaded tests (multiple clients hitting one server concurrently -- see Lecture 13), and, if your team's setup allows it, distributed tests across more than one machine.

16.3.6 Handling Failure: Server Down, Client Down

A client-server application that only works when both sides are healthy and the network is perfect is not finished. Your client must detect and respond sensibly when the server is unreachable or goes down mid-session -- not hang indefinitely or crash with an unhandled exception. Symmetrically, your server must detect and respond sensibly when a client disconnects unexpectedly -- not leak a thread or a socket resource, and not crash the rest of the server for other connected clients.

This is exactly Lecture 14's warning, at team-project scale

Lecture 14 built LibraryClient and LibraryServer with explicit try/catch around every socket operation, specifically because a network call can fail in ways a local method call never does. A capstone project that skips this is repeating the exact mistake that lecture's Section 14.6 called out by name.

16.4 Final Exam Format: Team Project, Open-Book

The Final is a team project of up to three students, open-book, evaluated on the deliverable described in Section 16.3 in its entirety: your components diagram, specification and interface documentation, protocol description and flowchart, bidirectional test cases, phase-by-phase documentation, and a working, demonstrable client-server application with disclosed, deliberate failure handling.

Academic integrity

By submitting this project, each team member affirms that the work submitted is their own team's genuine effort, that any external code, library, or reference used is disclosed rather than presented as original, and that each member's individually claimed contribution accurately reflects the work they actually did.

Criterion	What it looks for	Weight
Design	Components diagram, specification and interface documentation, and protocol/flowchart clarity.	20%
Implementation	A working client-server application matching the design, including the GUI, concurrency, and networking layers.	50%
Analysis & evaluation	Bidirectional test cases, test plans (stub-based, multithreaded, distributed where applicable), and disclosed failure-handling behavior.	25%
Conclusion	A clear, honest summary of what the team built, what worked, and what the team would do differently with more time.	5%

16.5 OOP in Practice: Profitina

About this box

As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina itself is, at its core, exactly this chapter's shape at production scale: a GUI-facing client talking over the network to a server that owns the real business logic and data, built from components with their own specifications, tested bidirectionally, and engineered from day one to detect and recover from a down dependency rather than to assume the network is always perfect. A capstone project that gets this chapter's four ingredients right — GUI, concurrency, networking, and a handful of well-named design patterns — has, in miniature, built the same shape of system this course's author builds professionally.

16.6 Chapter Summary

- This chapter introduced no new material — it is a capstone drawing together Lecture 11 (GUI & MVC), Lecture 13 (Concurrency), Lecture 14 (Networking), and Lecture 15 (Design Patterns) into one integrated project.
- A team of up to three students designs and builds a GUI-based, socket-based client-server application, on a topic of their choosing, demonstrated with a components diagram, interface documentation, a protocol description with a flowchart, and bidirectional test cases.
- Documentation spans four kinds: User-level, Performance criteria, Installation, and Troubleshooting — alongside Specification, Design, and Implementation phases.
- Failure handling is graded, not optional: both a down server and a down client must be detected and handled gracefully, on both sides, using disclosed exception handling.
- Grading weights Implementation heaviest (50%), followed by Analysis & Evaluation (25%), Design (20%), and Conclusion (5%).

Sixteen lectures ago, this course began with a single question: what changes when a program is organized around objects rather than procedures? Every chapter since has been one more answer to that question, in increasingly larger pieces — and this final project is the first time all of those pieces have to work together, at once, talking to each other across a network, exactly as real software does.

Appendix 16.A — Self-Check Checklist (Non-Graded)

Before your team submits, run the whole project through this checklist. It costs an afternoon and catches most of what graders flag most often in a capstone of this kind.

- Does the components diagram cover Server, Client, database/persistence connectivity, GUI, and testing/documentation, with each component owned by a named team member?
- Is there a written specification for every component another team member depends on — inputs, outputs, and failure guarantees — not just a Javadoc comment on a method signature?
- Does the protocol description include a flowchart, and does it unambiguously say whose turn it is to speak at every step, the way the Knock-Knock example does?
- Did the client's owner design test cases for the server, and did the server's owner design test cases for the client — not only tests each wrote for their own code?
- Does the test plan include stub-based tests, a multithreaded test with more than one concurrent client, and (if feasible) a distributed test across more than one machine?
- Does the client detect and handle a server that is down or goes down mid-session, without hanging or crashing uninformatively?
- Does the server detect and handle a client that disconnects unexpectedly, without leaking resources or affecting other connected clients?
- Do the four documentation deliverables (User-level, Performance, Installation, Troubleshooting) all exist, and would a teammate who joined today be able to run the system from them alone?

References

[1] This chapter's assessment format and grading rubric are modeled closely on this course's real Final Exam (EF234302 Object-Oriented Programming): a team project of up to three students, open-book, building a GUI-based, socket-based client-server application on a topic of the team's choosing, graded on Design (20%), Implementation (50%), Analysis & Evaluation (25%), and Conclusion (5%). Unlike the real Midterm (see Chapter 8's Reference [1]), whose recursive-tree and list-based content fell outside this rebuild's OOP scope and needed fresh substitute problems, the real Final's actual required content — a components diagram spanning Server/Client/GUI/database connectivity/testing, specification and interface documentation, a disclosed communication protocol illustrated with the same Knock-Knock example used above, bidirectional test-case design, phase-by-phase documentation, and deliberate server-down/client-down failure handling — maps directly onto exactly what this rebuild actually taught in Lectures 11, 13, 14, and 15, and is reused here largely as-is. Administrative specifics from the real document (submission emails, dates, file-naming conventions, and its own religiously-worded Academic Integrity Pledge) have been generalized or paraphrased for a textbook audience rather than reproduced verbatim.

[2] Oracle Corporation. "All About Sockets." The Java Tutorials. https://docs.oracle.com/javase/tutorial/networking/sockets/ (accessed August 2026).

[3] Oracle Corporation. "Lesson: Concurrency." The Java Tutorials. https://docs.oracle.com/javase/tutorial/essential/concurrency/ (accessed August 2026).