Object-Oriented Programming

First Edition

Irfan Subakti

PYTHON 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 Python 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 (attributes) and behavior (methods), and explain encapsulation as "hiding the data behind the behavior," Python-style. (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 Python interpreter and editor. (5) Read and explain every line of a minimal Python class and its accompanying driver script, and run it yourself.

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

OS	One-time install (Python 3.12+)	Run (identical everywhere)
Windows 11	winget install Python.Python.3.12 (or python.org installer)	python main.py (py main.py also works)
macOS	brew install python@3.12 (or python.org installer)	python3 main.py
Ubuntu/Linux	sudo apt install python3 python3-pip	python3 main.py

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, lists, 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 lists 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, abstract base classes, duck typing, 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 (a statically-typed, industry-standard OOP language) and once in Python (this edition — a dynamically-typed language whose OOP features are more lightweight but conceptually identical). This is the Python edition. A companion Java edition exists with the exact same 16-lecture roadmap and the exact same worked examples, translated into idiomatic Java — 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).

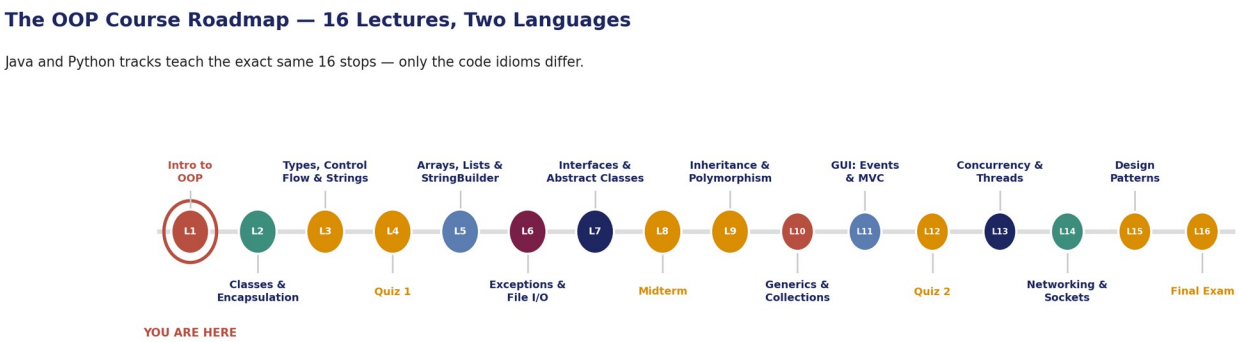


Figure 1.1 — The 16-lecture OOP roadmap. Both the Java and Python tracks visit the exact same 16 stops; only the code idioms differ. You are here: Lecture 1.

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 lists: one list of titles, one of authors, one of ISBNs, and one list of booleans 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 lists 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 on_loan list and setting an entry to True without ever checking whether that book was actually available, or forgetting to update times_renewed when a loan is renewed, or updating one list 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 lists, and every new function added to the program is one more place that guarantee can quietly break.

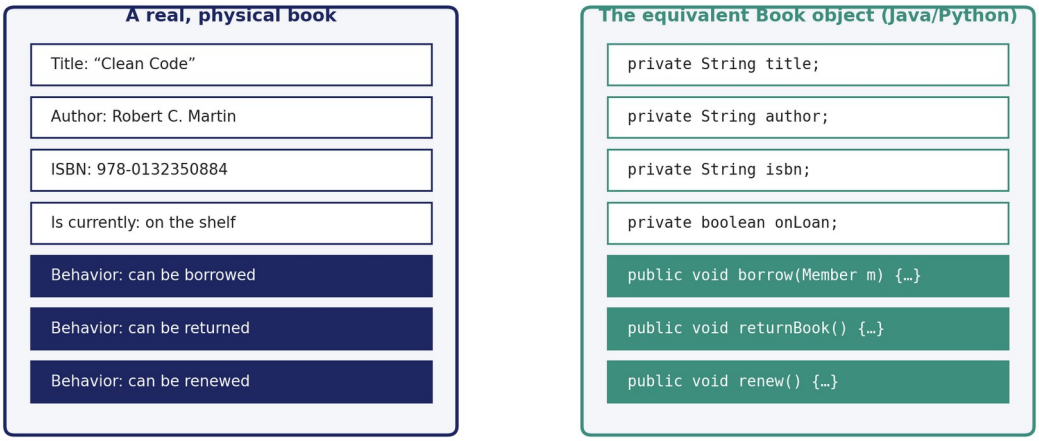
> **"Just be careful" does not scale**
>
> A common first reaction is: "the programmer just needs to be careful not to misuse the lists." This works for a 200-line student assignment written by one person in one sitting. It reliably fails for any program maintained by more than one person, touched more 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 conventions and tools — properties, name-mangling, type hints checked by tools like mypy — that make many classes of mistakes far easier to catch, 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 attributes 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, return_book, renew) that make sense to perform on it.

Encapsulation, previewed — Python style

Java marks a field private and the compiler enforces it. Python has no compiler-enforced private keyword; instead, a single leading underscore (as in _title, _on_loan in Figure 1.2's Book object) is a naming CONVENTION meaning "this belongs to the class's own internals — outside code should not touch it directly," and a @property decorator can additionally expose a controlled, read-only view. Python trusts programmers to respect that convention rather than having the interpreter refuse to compile a violation. This bundling-plus-hiding is still 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 abstract base classes and protocols (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 lists and a set of free functions that must all be given the right list and the right index every time. The object-oriented version on the right defines one Book class; each Book object owns its own title and _on_loan attribute, and by convention the only way to change _on_loan is to call a method that Book itself defines and can refuse.

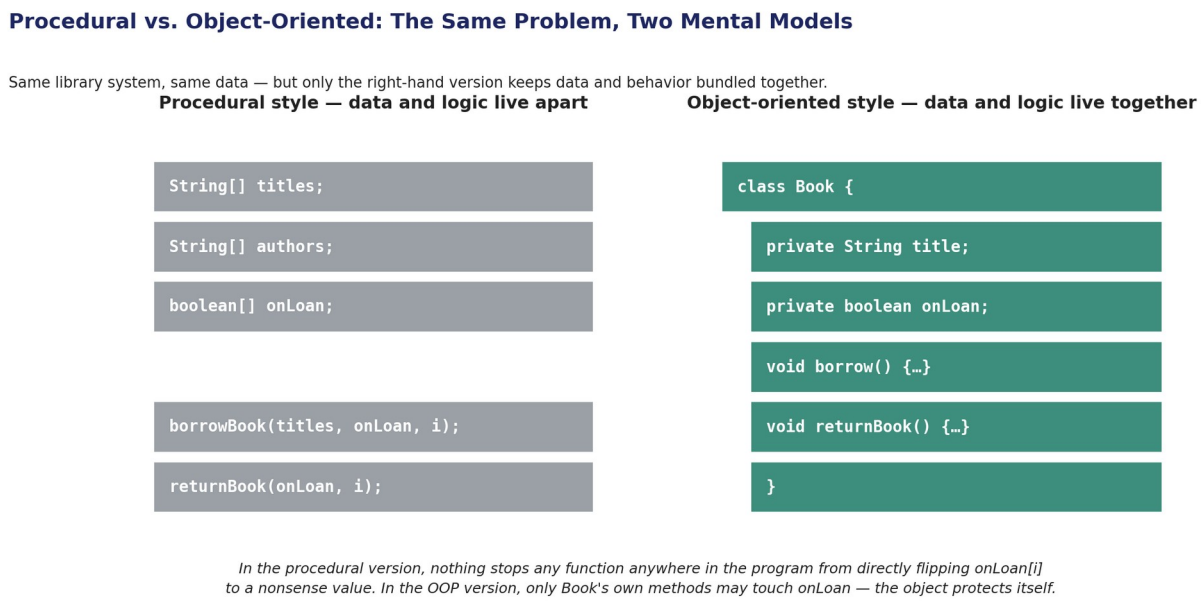


Figure 1.3 — The procedural version can be corrupted from anywhere in the program. The object-oriented version keeps its data and the logic that changes it in one place, by convention and by design.

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 (by underscore convention and @property) behind those methods so it can only change in valid ways.
- Abstraction via abstract base classes & protocols (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 attributes 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. Python's dynamic "duck typing" makes this especially natural.

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 (abstract base classes) until you first accept that an object's data should be hidden behind its own methods.

1.6 Setting Up Your Python Development Environment

Every example in this book is written for a current Python 3 release. As of August 2026, Python 3.14 is the latest stable release (first released October 2025), with active support continuing into 2027 and security-only support through 2030; Python 3.13 remains in active support as well. This book's own code was written against and run on Python 3.11, and everything shown runs unchanged on 3.12, 3.13, and 3.14, since none of this course's material depends on a feature removed or changed between those releases. Do not follow tutorials written for Python 2 — Python 2 reached its end of life in January 2020 and is not compatible with the syntax this course uses (for example, print() as a function, not a statement).

Tool	Recommendation	Why
Interpreter	CPython 3.12, 3.13, or 3.14 (python.org)	The reference implementation; free, and what nearly every tutorial and library assumes.
Editor — primary	Visual Studio Code (free) + the Python extension	The most widely used Python editor in industry today; excellent debugging, linting, and virtual-environment support.
Editor — alternative	PyCharm Community Edition (free)	A full-featured, Python-specific IDE with strong refactoring tools, a fully capable free alternative to VS Code.
Package/venv tool (later chapters)	venv (built-in) + pip	Introduced when a chapter needs a dependency beyond the standard library (e.g. pytest or a GUI toolkit in Lecture 11).

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

Every code listing in this book is plain .py source runnable with the standard CPython interpreter — it will run identically whichever editor (or no editor at all, from a terminal) you use to write it. Pick whichever tool your class, employer, or lab requires; nothing in this course depends on editor-specific project files.

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

```
$ python3 --version
Python 3.11.15
```

1.7 Worked Example: Your First Python 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

```
Attributes (leading underscore = 'internal, do not touch directly'):
  _title, _author, _isbn : str    (never change after creation)
  _on_loan               : bool   (starts False)
  _times_renewed          : int    (starts 0, resets on each new borrow)
Methods (the conventional way outside code affects this object):
  borrow()      -> bool   returns False if already on loan
  return_book() -> None
  renew()       -> bool   returns False if not on loan or already
                          renewed twice
  __str__()     -> str    human-readable summary, for printing
```

Phase 4 — Analyze

```
Correctness argument: _on_loan can only become True inside borrow(),
and only when it was False; it can only become False inside
return_book(). _times_renewed can only increase inside renew(), and
only up to 2, and is reset to 0 every time borrow() succeeds. As long
as calling code respects the underscore convention (Section 1.3), no
code elsewhere in the program changes these attributes except through
these four methods.
```

Phase 5 — Implement

```
class Book:
    def __init__(self, title: str, author: str, isbn: str) -> None:
        self._title = title
        self._author = author
        self._isbn = isbn
        self._on_loan = False
        self._times_renewed = 0

    def borrow(self) -> bool:
        if self._on_loan:
            return False
        self._on_loan = True
        self._times_renewed = 0
        return True

    def return_book(self) -> None:
        self._on_loan = False

    def renew(self) -> bool:
        if not self._on_loan or self._times_renewed >= 2:
            return False
        self._times_renewed += 1
        return True

    def __str__(self) -> str:
        status = (
            f"ON LOAN (renewed {self._times_renewed}x)"
            if self._on_loan else "on the shelf"
        )
        return (
            f'"{self._title}" by {self._author} '
            f'[ISBN {self._isbn}] - {status}'
        )
```

A class by itself does nothing until something creates an object from it — Python calls this instantiation, done simply by calling the class name like a function. The following driver script (a separate module holding a main() function) 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 return_book(), ships as Code/Python/Chapter01/library_demo.py alongside this book, and its full output is what Appendix 1.A verifies:

```python
from pustaka import Book

def main() -> None:
    clean_code = Book("Clean Code",
        "Robert C. Martin", "978-0132350884")

    print(clean_code)
    clean_code.borrow()
    print(clean_code)
    again = clean_code.borrow()
    print(f"Second borrow allowed? {again}")

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

Trace it yourself before reading on

What does the program above print on its fourth printed line? Walk through it: the first print shows the freshly-constructed book (on the shelf); borrow() then flips _on_loan to True; the second print 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 _on_loan cannot become True a second time without a return_book() in between. If your trace disagrees with "Second borrow allowed? False", re-read borrow()'s single if statement.

1.8 Running Your Program

Python is (usually) an interpreted language: the CPython interpreter reads your .py source and executes it directly, without a separate compile-then-run step the way Java requires. Running python3 library_demo.py both loads pustaka.py (via the import statement) and executes library_demo.py's own code in one step:

```
$ python3 library_demo.py
```

Before you actually run a script, it is good practice to at least check it for syntax errors with python3 -m py_compile, which parses the file (and writes a cached bytecode file) without executing it — this is the closest Python equivalent to Java's separate javac compile step, and this course's Appendix verification logs use it as a first check before actually running each example:

```
$ python3 -m py_compile pustaka.py library_demo.py
$ echo $?
0
```

If you are using an editor such as VS Code or PyCharm, the editor runs python3 your_file.py for you behind a "Run" button — understanding what actually happens underneath that button will make editor 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 — the exact same language as this book's code, using the exact same encapsulation idiom this chapter introduces. 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, following the same underscore-plus-property convention shown in Section 1.7, rather than exposing that score as a bare attribute 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 Python (or Java) 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 lists have no single place enforcing that the data stays sensible — any function anywhere can corrupt it.
- An object bundles data (attributes) with the behavior (methods) that is allowed to change that data; encapsulation in Python means hiding attributes behind a leading underscore and exposing controlled access via @property, since Python has no compiler-enforced private keyword.
- 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.
- Python is interpreted: CPython runs .py source directly. Use Python 3.12, 3.13, or 3.14 (all current as of 2026), and VS Code or PyCharm as your editor.
- 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 _times_renewed attribute uses a leading underscore. Write one sentence explaining what convention this signals to other programmers, and why Python trusts programmers to respect it rather than having the interpreter enforce it.
3. Modify (on paper, or actually in your editor) the Book class to add a method extend_due_date(self, days: int) that is only allowed to succeed if the book is currently on loan. Where in the class must this rule be enforced, and why?
4. Figure 1.3 shows a procedural version with four parallel lists. Suppose a new field, due_date, 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 — Execution Verification Log

The complete pustaka.py and library_demo.py files (Code/Python/Chapter01/, provided alongside this book — a superset of the shortened excerpt in Section 1.7, exercising a second book plus renew() and return_book() as well) were syntax-checked and executed on CPython 3.11.15 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ python3 -m py_compile pustaka.py library_demo.py
(no errors)

$ python3 library_demo.py
--- 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] Python Software Foundation. "Status of Python Versions." https://devguide.python.org/versions/ (accessed August 2026) — confirms Python 3.14 as the latest stable release (October 2025), 3.13 still in active support.

[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] Microsoft. "Visual Studio Code — Python." https://code.visualstudio.com/docs/languages/python (accessed August 2026).

[5] JetBrains. "PyCharm." https://www.jetbrains.com/pycharm/ (Community Edition, accessed August 2026).

CHAPTER 2

Anatomy of a Class: Attributes, 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 — attributes, constructors, methods — and state what each is for. (2) Write __init__, and explain what "self" refers to inside one. (3) Use a default parameter value in __init__ to let one constructor cover cases that Java would need constructor overloading for. (4) Write @property getters that expose an object's attributes for reading without allowing outside code to corrupt them. (5) Name Java's four access modifiers, and explain what Python uses instead of compiler-enforced access control. (6) Extend the Chapter 1 Book class with a new attribute, a richer constructor, and full property coverage, then syntax-check, 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 (attributes) with the behavior (methods) allowed to change that data, and hides the attributes behind that behavior. You saw one class, Book, with three main attributes and four methods, and you 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 — attributes, 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.

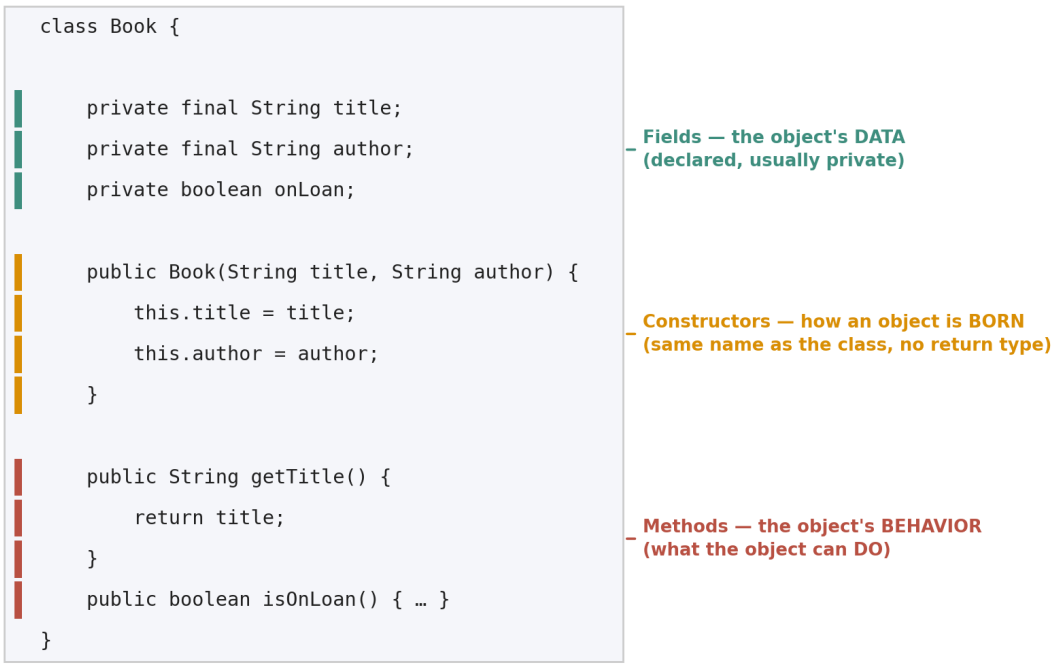


Figure 2.1 — Every class you write has (at most) these three kinds of members, in this conventional order: fields/attributes, then a constructor, then methods. (Shown here in Java syntax; Python's own equivalent shape appears throughout this chapter.)

2.2 Attributes: An Object's Data, Formally

An attribute is a name bound to a value on self inside a class's methods (most commonly inside __init__) — it is where an object's data lives. Chapter 1's Book set five attributes inside __init__: _title, _author, _isbn, _on_loan, and _times_renewed. Every object built from the Book class gets its own independent copy of these five attributes; two different Book objects never share the same _title value, even though both are set by the exact same line of source code inside __init__ (Figure 2.1).

Notice that _title, _author, and _isbn are never reassigned anywhere outside __init__ in this course's Book class — by convention, they are meant to be effectively immutable once the object is built, the same intent Java expresses with the final keyword. Python has no final keyword and no compiler that enforces "this cannot be reassigned"; the leading underscore plus the absence of a matching @x.setter is the entire mechanism, and it depends on other programmers respecting the convention rather than the interpreter refusing to compile a violation. _on_loan and _times_renewed are deliberately designed to change, because their whole purpose is to track the book's state as it is borrowed, renewed, and returned.

Java's private + final, side by side with Python's convention

Java marks title, author, and isbn private final: private is enforced by the compiler (code outside the class cannot even attempt to read or write the field directly), and final is a second, independent, compiler-enforced guarantee that the field cannot be reassigned after construction. Python's _leading_underscore signals the same INTENT as private, but the interpreter enforces none of it — there is no Python equivalent of final at all. This is not a shortcoming particular to Python; it reflects a broader design philosophy ("we are all consenting adults here") that trusts programmer discipline over compiler enforcement, discussed further in Section 2.6.

2.3 Constructors: How Objects Are Born

A class by itself is only a blueprint; nothing exists until code calls the class name like a function, and that call always runs __init__. 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 attributes set and others simply missing.

In Python, __init__ is a method with two unusual properties: its first parameter is always self, referring to "the object currently being built," and it returns None implicitly (an __init__ that returns anything else raises a TypeError). Every other parameter is supplied by the caller. Inside __init__, self.title = title unambiguously means "the attribute belonging to this object gets the value of the parameter" — there is no naming collision to resolve the way Java's this. resolves one, because self. is always required to reach an attribute; a bare title inside __init__ can only ever refer to the parameter, never to something on the object.

Forgetting self. is Python's version of a classic first bug

If you write title = title inside __init__ instead of self._title = title, Python does not raise an error — it creates (or reassigns) a plain local variable named title that vanishes the moment __init__ returns, and self never gets a _title attribute at all. The very next line that tries to read self._title raises AttributeError, but often not until some other method is called much later, which can make the root cause hard to trace back to the missing self.. Java's equivalent mistake (Section 2.3 of the Java edition) fails just as silently at the point of assignment, but for the opposite structural reason — Java has the naming collision Python's self. requirement prevents entirely.

2.4 One Constructor, Default Arguments — Python's Answer to Overloading

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 handles this with constructor overloading — multiple constructors, distinguished by their parameter lists (see the Java edition's Section 2.4 and Figure 2.2 for the full picture). Python's method signatures do not support true overloading — there is exactly one __init__ per class — but a default parameter value covers exactly the same use case with less code.

def __init__(self, title, author, isbn, publication_year=0): gives publication_year a default value of 0, meaning "not recorded." Calling Book(t, a, i) and Book(t, a, i, 2008) both work, and both run the exact

same __init__ body — there is no second constructor to keep in sync, no this(...)-style delegation to learn, and no risk of the two "versions" drifting apart the way Java's two overloaded constructors could if someone edited one and forgot the other. This is one of the places where Python's dynamic flexibility genuinely simplifies a pattern Java needs a dedicated language feature (overloading) to express.

> **Constructor overloading, for comparison (Java)**
>
> Java would write this as TWO constructors: Book(title, author, isbn), whose one-line body is this(title, author, isbn, 0);, and Book(title, author, isbn, publicationYear), which does the actual five-attribute assignment. The first delegates to the second via constructor chaining, so the assignment logic exists in exactly one place. Python's single __init__ with a default argument reaches the same guarantee — one place the assignment happens — through the language's parameter-default feature instead of a chaining call.

2.5 Properties: Controlled Read Access

Hiding an attribute behind a leading underscore (Section 2.2) solves the write-side of encapsulation — nothing outside the class corrupts the attribute by convention. But it also, as an unavoidable side effect, discourages the outside world from simply reading the attribute'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 @property is a small method, decorated so it can be READ exactly like a plain attribute (book.title, not book.title()), whose entire job is to return one attribute's current value.

A property getter is deliberately the simplest possible kind of method: besides self, it takes no parameters, and its body is a single return statement. Because @property lets calling code write book.title instead of book.get_title(), Python code that uses properties looks exactly like code that uses plain public attributes — the caller cannot tell, from the call site alone, whether title is a raw attribute or a computed property, which is precisely the point: you can start with a plain public attribute and convert it to a @property later, without changing a single line of calling code.

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

Python @property	Java getter	Returns

A property returning a str or int is still safe, even without "copying" it

A natural worry: if title just returns self._title directly, can't the caller who receives it turn around and corrupt Book's internal state through that returned reference? For str, int, and bool specifically, no — these types are immutable in Python, meaning there is no operation on a str or an int that changes 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 internal attribute. (This question becomes far more interesting once a property returns a mutable type, such as a list of loans — a problem this course revisits directly in Chapter 10, Generics & Collections.)

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

Section 2.2 and Chapter 1 have both used the leading-underscore convention repeatedly without fully cataloging what Java offers instead. Java defines four levels of compiler-enforced visibility a field, constructor, or method can be given; Python offers exactly one naming convention and trusts the programmer for the rest (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.

Java's private is the narrowest level: visible only inside the class's own body, and enforced by the compiler. package-private (Java's default, when no keyword at all is written) opens visibility to every other class in the same package. protected extends that 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 Java uses for every getter and every behavior method. Python has no equivalent of package-private or protected at the language level at all — a single leading underscore covers everything Python considers "internal," and a double leading underscore (name-mangling, e.g. self.__secret) exists mainly to avoid accidental name clashes in subclasses, not to enforce true privacy; it can still be reached from outside the class if you know the mangled name.

> **No underscores anywhere defeats the entire point of Chapter 1**
>
> It is technically legal to name every attribute without a leading underscore and delete every property — nothing in Python prevents this, and no error is ever raised. Doing so throws away everything Chapter 1 argued for: with every attribute a plain public name, any code anywhere in the program can write book.on_loan = 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 underscore character by itself; it is underscore-prefixed attributes deliberately paired with a small, curated set of public methods and properties.

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 _publication_year attribute (via a default constructor argument), and a @property for every attribute.

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 (publication_year), plus a
controlled READ channel (@property) 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 attribute:
  _publication_year : int  (0 means "not recorded")
Constructor (ONE __init__, default argument covers both cases):
  __init__(self, title, author, isbn, publication_year=0)
Properties (read-only; the ONLY read channel for underscored
attributes):
  title, author, isbn, publication_year, is_on_loan,
  times_renewed   -- all take only self, single-return
```

Phase 4 — Analyze

```
Correctness argument: _publication_year is set exactly once, inside
__init__, exactly like _title, _author, and _isbn -- no other
method in this course's Book class reassigns it. Every property
returns an immutable type (str, int, or bool), so no caller who
reads a property's value can reach back in and mutate Book's
internal state through it. There is only ONE constructor, so there
is no risk of two constructors drifting out of sync the way two
overloaded Java constructors could.
```

Phase 5 — Implement

```
class Book:
    def __init__(self, title: str, author: str, isbn: str,
                 publication_year: int = 0) -> None:
        self._title = title
        self._author = author
        self._isbn = isbn
        self._publication_year = publication_year
        self._on_loan = False
        self._times_renewed = 0

    # borrow(), return_book(), renew() -- unchanged from Chapter 1

    @property
    def title(self) -> str:
        return self._title

    @property
    def author(self) -> str:
        return self._author

    @property
    def isbn(self) -> str:
        return self._isbn

    @property
    def publication_year(self) -> int:
        return self._publication_year

    @property
    def is_on_loan(self) -> bool:
        return self._on_loan

    @property
    def times_renewed(self) -> int:
        return self._times_renewed
```

The driver script below creates two Book objects, one supplying a known publication year and one relying on the default value (unrecorded, defaulting to 0), then exercises several properties. It is shown here in full, exactly as it appears in Code/Python/Chapter02/library_demo.py, and its complete output is exactly what Appendix 2.A verifies:

```
from pustaka import Book

def main() -> None:
    clean_code = Book("Clean Code", "Robert C. Martin",
                      "978-0132350884", 2008)
    pragmatic = Book("The Pragmatic Programmer", "Hunt & Thomas",
                     "978-0135957059")

    print(clean_code)
    print(pragmatic)

    print(f"Clean Code author: {clean_code.author}")
    print(f"Pragmatic Programmer year: {pragmatic.publication_year} "
          f"(0 = unrecorded)")

    clean_code.borrow()
    print(clean_code)

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

Trace it yourself before reading on

pragmatic is built without a fourth argument, so publication_year falls back to its default. What does pragmatic.publication_year return, and what does the __str__() line print for pragmatic's year, given that? Trace it: since no fourth argument was passed, publication_year defaults to exactly 0, and __str__() (Section 2.7's Phase 5, extended further in the full source file) is written to omit the year bracket entirely whenever _publication_year is 0, rather than printing the misleading number 0 itself.

2.8 Running Your Program

Nothing changes about the run process Chapter 1, Section 1.8 introduced: python3 -m py_compile checks both files for syntax errors, and python3 library_demo.py both loads pustaka.py (via the import statement) and runs library_demo.py's own code.

```
$ python3 -m py_compile pustaka.py library_demo.py
$ python3 library_demo.py
```

If you extend pustaka.py yourself while working through this chapter's Review Questions, running python3 -m py_compile first is worth the extra second — it catches a typo or an indentation mistake before you have to untangle it from a traceback produced by actually running the (still broken) script.

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 __init__ methods mirror exactly the pattern this chapter teaches: a scan-result class, for instance, is built once per completed AI-visibility scan, with every attribute 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 _publication_year attribute 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 @property, never by reaching directly into the object's underscore-prefixed attributes, 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 product code shows the exact convention this chapter teaches at production scale: every one of Profitina's own classes exposes its data exclusively through @property getters, not bare attribute access, and every internal attribute uses the leading-underscore convention throughout — the same convention this chapter's Book class follows, just with far more at stake than a library catalog.

2.10 Chapter Summary and Key Takeaways

- A class has (at most) three kinds of members: attributes (the object's data), a constructor (how the object is built), and methods (the object's behavior).
- __init__'s job is to bring one object into existence in a valid state; self. is always required to reach an attribute, so Python has no equivalent of Java's this.-vs-bare-name naming collision.
- Python has no constructor overloading — there is exactly one __init__ per class — but a default parameter value covers the same use case Java needs multiple constructors for, with the assignment logic living in exactly one place either way.
- A @property is a small method that returns one attribute's value while looking, to calling code, exactly like a plain attribute read — restoring read access without reopening write access.
- Java has four compiler-enforced access levels — private, package-private, protected, public; Python has exactly one convention (leading underscore) and trusts the programmer for everything else.
- Naming every attribute without a leading underscore defeats the purpose of encapsulation established in Chapter 1 — underscore-prefixed attributes deliberately paired with a small set of public methods and properties 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 attribute, a richer constructor, six properties — 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. Python's single __init__ with a default argument and Java's two overloaded constructors both guarantee the assignment logic lives in exactly one place, but they get there by different mechanisms. Explain, in your own words, what each language's mechanism actually is.
2. What would happen — be specific about what error appears and when — if you wrote publication_year = publication_year (instead of self._publication_year = publication_year) inside Book's __init__?
3. is_on_loan is a property name without a get_ prefix, unlike a hypothetical get_is_on_loan. Write one sentence explaining why Python properties conventionally drop verb prefixes entirely, unlike Java's get-/is- getter convention.
4. Suppose a publishers property were added that returns a mutable list of past publishers rather than a single str. Explain why Section 2.5's INSIGHT box's safety argument would no longer fully apply, and what could go wrong.
5. Write a remaining_renewals property for Book that returns how many renewals are still available (the maximum of 2, minus times_renewed). Should it be a @property or a regular method taking arguments, and does adding it require touching any existing attribute's naming convention? Justify your answer.

Appendix 2.A — Execution Verification Log

The complete pustaka.py and library_demo.py files (Code/Python/Chapter02/, provided alongside this book) were syntax-checked and executed on CPython 3.11.15 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ python3 -m py_compile pustaka.py library_demo.py
(no errors)

$ python3 library_demo.py
"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] Python Software Foundation. "Classes." https://docs.python.org/3/tutorial/classes.html (accessed August 2026).

[2] Python Software Foundation. "Status of Python Versions." https://devguide.python.org/versions/ (accessed August 2026) — confirms Python 3.14 as the latest stable release (October 2025), 3.13 still in active support.

[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] Microsoft. "Visual Studio Code — Python." https://code.visualstudio.com/docs/languages/python (accessed August 2026).

[5] JetBrains. "PyCharm." https://www.jetbrains.com/pycharm/ (Community Edition, accessed August 2026).

CHAPTER 3

Built-in 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: Python's dynamically-typed built-in types, the two ways to branch (if/elif/else and the match statement), Python's own loop forms, why strings compare 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 Python's core built-in types (int, float, bool, str, complex) and state which ones are mutable. (2) Write if/elif/else chains and a Python 3.10+ match statement to encode branching logic, and explain why match is a statement, not an expression. (3) Choose correctly among for and while loops, and explain why Python has no C-style for or do-while loop. (4) Explain why Python's == already compares string content, and why relying on is for strings is never safe even when it happens to work. (5) Use print debugging, pdb 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 syntax-check, run, and verify it yourself.

3.1 Picking Up Where Chapter 2 Left Off

Chapter 2 opened up the Book class and looked at its three kinds of members: attributes, __init__, and properties. Every method you wrote in Chapters 1 and 2 -- borrow(), renew(), every @property -- 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 built-in types, decisions (if this, then that), repetition (do this several times), and text (str). 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.

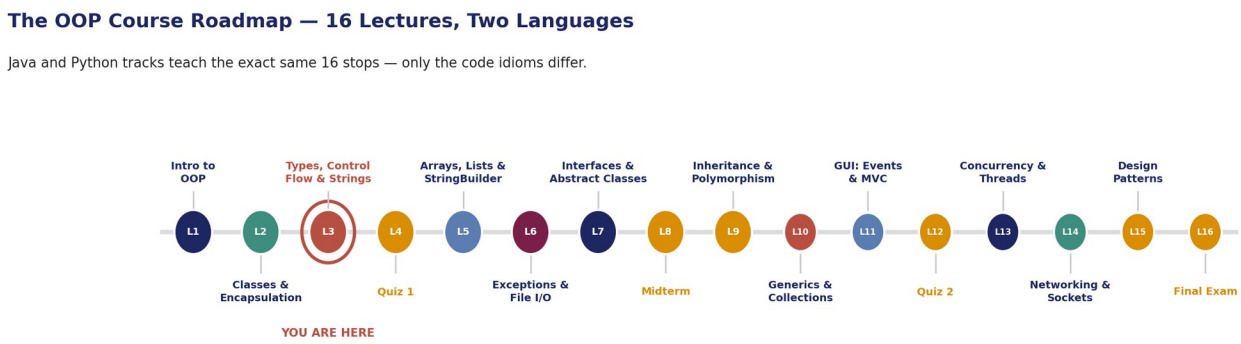


Figure 3.1 — The 16-lecture OOP roadmap. This chapter is Lecture 3, the last stop before Quiz 1 (Lecture 4) checks Lectures 1 through 3 together.

3.2 Python's Built-in Types

Every value in Python is an object -- even an int or a bool -- but a small set of built-in types cover the overwhelming majority of everyday code: int, float, bool, str, and complex. Unlike Java's eight primitives, none of these has a fixed size Python exposes to you: an int can grow arbitrarily large without overflowing (Python allocates more memory automatically), and there is no distinct byte, short, or long to choose among. Book already uses two of these: int (publication_year, times_renewed) and bool (on_loan, and every == comparison's result) (Figure 3.2).

Python's Built-in Types: Dynamic, Not Primitive

Python has no separate "primitive" category -- every value, including numbers and booleans, is an object.

Type	Example	Mutable?	Java's closest analog
int	42, -7, 10**20	No	int / long, but unbounded
float	3.14, 2.0	No	double (no separate float type)
bool	True, False	No	boolean (but bool IS an int subclass)
str	"Clean Code"	No	String
complex	3+4j	No	(no built-in analog)

int has no fixed size or overflow point -- it grows automatically to hold any integer. bool is actually a subclass of int (True == 1 is True) -- a genuine language difference, not a simplification.

Figure 3.2 — Python's core built-in types, with an example, mutability, and each one's closest Java analog. bool is technically a SUBCLASS of int, a detail with real consequences (Section 3.2's callout).

Two practical rules cover the overwhelming majority of real code. First, default to int for whole numbers and float for numbers with a fractional part; Python has no separate double to choose -- float is always a 64-bit IEEE 754 value, the same underlying representation as Java's double. Second, str holds text of any length and is not interchangeable with a single character the way Java's char is -- Python has no separate character type at all; a one-character str is still simply a str.

bool being an int subclass is not just trivia

Because bool is a subclass of int, True == 1 and False == 0 both evaluate to True in Python, and True + True actually evaluates to 2. This occasionally surprises code that expects a strict True/False comparison to fail against a numeric 1 or 0 -- it will not. And exactly as in Java, comparing float values with == is a classic silent bug: 0.1 + 0.2 == 0.3 evaluates to False in Python for the identical IEEE 754 reason Java has the same problem. Never compare float values for exact equality; compare abs(a - b) against a small tolerance instead, or use math.isclose(a, b).

3.3 Control Flow: if/elif/else and match

An if statement's condition is evaluated for truthiness -- Python is more permissive here than Java: 0, 0.0, "", [], and None are all falsy, and any other value is truthy, so an int or a str CAN appear directly in a condition without an explicit comparison, unlike Java. Chaining elif lets a decision cascade through

several mutually exclusive possibilities, ending in an optional else that catches everything the earlier branches did not.

Python 3.10 introduced match, a second way to make the same kind of decision when every branch compares one value against a small set of specific cases. A match statement's case arms are checked top to bottom, and case _: is the catch-all wildcard, playing the same role Java's default does inside a switch. The critical difference from Java's switch expression: Python's match is always a STATEMENT, never an expression -- match cannot itself be returned or assigned; each case's body must contain its own return or assignment (Figure 3.3).

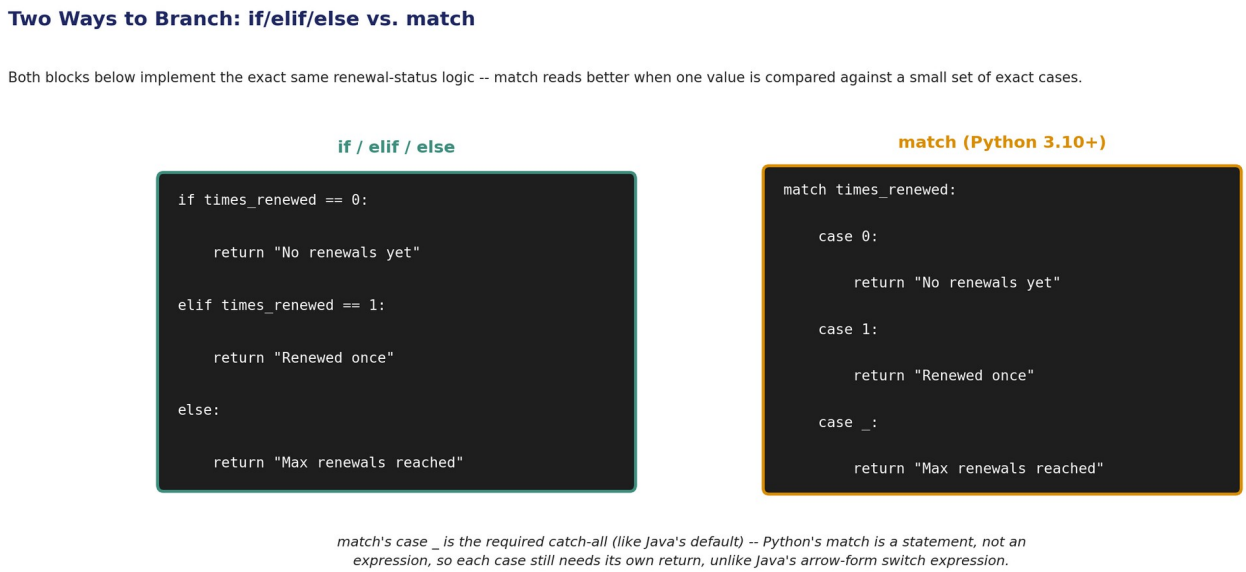


Figure 3.3 — The same renewal-status decision, written as if/elif/else and as a Python 3.10+ match statement. Both are correct; match communicates "one value, several specific cases" more directly, but unlike Java's switch expression it is a STATEMENT and returns nothing itself.

> **When to prefer match over if/elif/else**
>
> Reach for match when every branch tests the same single value against specific cases (an int, a str, or a structural pattern) -- exactly renewal_status_label()'s situation in Section 3.7. Prefer if/elif/else when branches test different values, or test ranges/inequalities rather than exact-match cases -- exactly calculate_fine()'s situation, which compares days_overdue against thresholds (<= 7, <= 30), not exact values match's simple case forms cannot express directly without an additional guard clause.

3.4 Loops: for and while

Python offers exactly two loop keywords, and Chapters 1-2 already used one of them (for x in some_list:) without naming it formally. Python has no C-style for (initialization; condition; update) loop at all, and no do-while loop -- for always iterates over an ITERABLE (a list, a string, a range, or anything else Python knows how to step through one item at a time), never a bare counter with a manually-written condition.

range(5) produces the integers 0 through 4, so for i in range(5): runs its body exactly five times -- the closest Python equivalent to Java's counting for loop, but expressed as iterating over a sequence of integers rather than manipulating an index directly. while condition: checks its condition before every

iteration, including the first -- it may run zero times if the condition starts out false, exactly like Java's while. Because Python has no do-while, a loop that must run at least once is written as while True: with an explicit break inside the body once the exit condition is met.

> **Iterating a list while modifying it is a classic silent bug**
>
> for item in my_list: my_list.remove(item) silently skips elements, because Python's for loop tracks a numeric position into the list, and removing an item shifts every later element one position earlier without the loop's position moving back to compensate. Iterate over a COPY (for item in my_list[:]: or for item in list(my_list):) whenever the loop body may add to or remove from the very list being iterated.

3.5 Strings: Value Comparison and Identity

A Python str, once created, can never be changed -- every str method that appears to modify a string (upper(), strip(), replace(...)) actually returns a brand-new str, leaving the original untouched. This is str immutability, the same property Java's String has, but Python resolves the comparison question far more simply than Java does: == always compares CONTENT for str (and for every other built-in type), never identity. There is no Python equivalent of Java's .equals() to remember, and no equivalent trap of comparing content with the wrong operator -- == already does the correct thing for strings, lists, and every other common type.

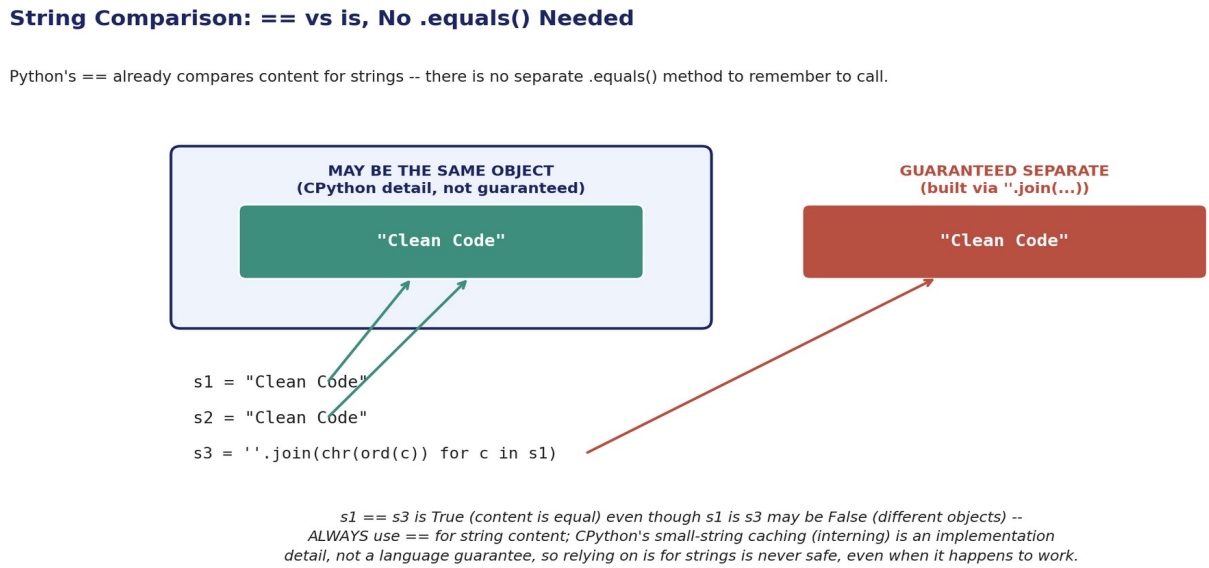


Figure 3.4 — s1 and s2, both str literals with the same text, MAY happen to be the same object under CPython's small-string interning -- but this is an implementation detail, never a language guarantee. s3, built via ''.join(...), is GUARANTEED to be a separate object, yet s1 == s3 is still True, because == always compares content.

is, Python's identity operator, answers a different question entirely: "are these two names bound to the exact same object in memory?" CPython (the reference implementation) happens to reuse ("intern") small string literals and small integers for efficiency, so "exit" is "exit" can print True in practice -- but this is a CPython implementation detail that the Python language specification does not guarantee, can differ across Python implementations, and can change between versions. Relying on is for strings is never safe, even on the rare occasion it appears to work.

is for strings: works by accident, breaks by design

isbn_from_scanner = "".join(chr(ord(c)) for c in isbn_from_catalog) builds a string with identical content to isbn_from_catalog through a genuinely separate object -- no interning applies to a string assembled this way. isbn_from_catalog == isbn_from_scanner is True (content matches); isbn_from_catalog is isbn_from_scanner is False (different objects). Always use == to compare string content in Python; reserve is exclusively for comparing against the singleton None (if x is None:), never for comparing two strings, two numbers, or two other ordinary values.

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
print() 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
pdb / breakpoint()	Following a program's exact execution path and inspecting many variables at once, without editing the source	Requires learning pdb's own command set (n, s, c, p, l); overkill for a one-value check
assert statements	Documenting and checking an internal invariant that should NEVER be false if the code is correct	Removed entirely when Python runs with the -O flag -- must never be relied on to run in every environment

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 first_name is not None is appropriate to document "this can never be None if my own code is correct" -- and because Python's -O flag strips every assert statement from the compiled bytecode entirely, 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 whenever -O is used. User input and other untrusted data belong behind an explicit if check (or, from Chapter 6 onward, a raised 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: calculate_fine(self, days_overdue: int) -> float, demonstrating built-in int/float arithmetic and an if/elif/else tier structure, and renewal_status_label(self) -> str, rewriting the exact same three-way decision already familiar from times_renewed as a match statement.

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 match
statement.
```

Phase 2 — Abstract

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

Phase 3 — Design

```
PHASE 3 -- DESIGN

New methods (no new attributes -- both read existing state or a
parameter):
  calculate_fine(self, days_overdue: int) -> float
    days_overdue <= 0            -> 0.00
    days_overdue in 1..7         -> days_overdue * 0.50
    days_overdue in 8..30        -> 3.50 + (days_overdue-7)*1.00
    days_overdue > 30            -> 50.00 (capped)
  renewal_status_label(self) -> str
    match self._times_renewed: case 0/1/_ -> ...
```

Phase 4 — Analyze

```
Correctness argument: calculate_fine'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 days_overdue grows, matching the requirement.
renewal_status_label's match statement is exhaustive by
construction (case _ covers every value not explicitly listed),
so it can never fail to return a value the way an incomplete
if-chain silently could.
```

Phase 5 — Implement

```
# attributes, __init__, and properties: unchanged from Chapter 2

def calculate_fine(self, days_overdue: int) -> float:
    if days_overdue <= 0:
        return 0.0
    elif days_overdue <= 7:
        return days_overdue * 0.50
    elif days_overdue <= 30:
        return 3.50 + (days_overdue - 7) * 1.00
    else:
        return 50.00

def renewal_status_label(self) -> str:
    match self._times_renewed:
        case 0:
            return "No renewals yet"
        case 1:
            return "Renewed once"
        case _:
            return "Max renewals reached"
```

The driver below demonstrates both new methods, loops over a list of sample overdue-day counts with a for loop (Section 3.4), and closes with the == vs is comparison Section 3.5 warned about, deliberately constructing a non-interned str with "".join(...) to make the difference visible. It is shown here in full, exactly as it appears in Code/Python/Chapter03/library_demo.py, and its complete output is exactly what Appendix 3.A verifies:

```
clean_code = Book("Clean Code", "Robert C. Martin",
                  "978-0132350884", 2008)
print(clean_code)

clean_code.borrow()
clean_code.renew()
print(f"Renewal status: {clean_code.renewal_status_label()}")

overdue_days_samples = [0, 3, 15, 45]
for days in overdue_days_samples:
    fine = clean_code.calculate_fine(days)
    print(f"Overdue {days:2d} day(s) -> fine: {fine:.2f}")

isbn_from_catalog = "978-0132350884"
isbn_from_scanner = "".join(
    chr(ord(c)) for c in isbn_from_catalog)
print(f"== comparison: {isbn_from_catalog == isbn_from_scanner}")
print(f"is comparison: {isbn_from_catalog is isbn_from_scanner}")
```

Trace it yourself before reading on

After borrow() and one renew() call, times_renewed is 1 -- what does renewal_status_label() return? Trace the fine tiers: 0 days is exactly 0.00 (the boundary belongs to the first branch, days_overdue <= 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 isbn_from_scanner, built with "".join(...), is guaranteed to be a different object from isbn_from_catalog -- so what does == print, and what does is print?

3.8 Running Your Program

Nothing changes about the process Chapters 1-2 introduced: python3 -m py_compile checks both files for syntax errors, and python3 library_demo.py both imports pustaka.py and runs library_demo.py's own code. The match statement requires no special flag on any Python from 3.10 onward (this course's CPython 3.11-3.14 baseline is well past that threshold).

```
$ python3 -m py_compile pustaka.py library_demo.py
$ python3 library_demo.py
```

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. See the companion document "Profitina: A Non-Confidential Technical Overview" for the full picture.

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 calculate_fine 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 exactly the value-based string comparison Section 3.5 describes -- never on comparing string identity with is, which this same section shows is a different, easily-confused question, one the Java track's Chapter 3 (Section 3.5) has to solve with .equals() instead.

3.10 Chapter Summary and Key Takeaways

- Python's core built-in types -- int, float, bool, str, complex -- have no fixed size Python exposes to you; bool is technically a subclass of int, with real arithmetic consequences.
- if/elif/else and match both encode branching; match is a STATEMENT (unlike Java's switch expression) and communicates "one value, several specific cases" directly when it applies.
- Python has exactly two loop keywords, for and while, and no C-style counting for loop or do-while; for always iterates an iterable, never a bare manually-incremented counter.

- Strings are immutable; == always compares content in Python (no .equals() needed), while is compares identity and is never safe to rely on for strings, even when CPython's interning happens to make it appear to work.
- print() debugging, pdb/breakpoint(), and assert each fit different situations; bisection localizes an unknown bug quickly; assert documents your OWN code's invariants, is stripped entirely under -O, and is never a substitute for validating untrusted input.
- Book gained calculate_fine(int) -> float and renewal_status_label() -> str this chapter, adding no new attributes -- both methods are pure functions of state Book already had.

Every tool this chapter introduced -- built-in 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 calculate_fine's if/elif/else chain as nested conditional expressions (a if condition else b). Is the result more or less readable than the if/elif/else version? Justify your answer.
2. Explain, precisely, why 0.1 + 0.2 == 0.3 is False in Python, and state the correct way to compare two float values for "close enough" equality.
3. What is printed by print("exit" is "exit")? What is printed by print("exit" is "".join(["e", "x", "i", "t"]))? Explain the difference using Figure 3.4, and why relying on either result would be a mistake.
4. Write a short loop-based method count_overdue_tiers(self, days_list: list[int]) -> int that returns how many days in the list fall into the 8-30 day fine tier. Which loop form (for or while) fits best, and why?
5. Section 3.6 says assert should never validate untrusted input. Rewrite this line so it correctly rejects a negative days_overdue argument to calculate_fine even when Python runs with -O: assert days_overdue >= 0

Appendix 3.A — Execution Verification Log

The complete pustaka.py and library_demo.py files (Code/Python/Chapter03/, provided alongside this book) were syntax-checked and executed on CPython 3.11.15 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ python3 -m py_compile pustaka.py library_demo.py
(no errors)

$ python3 library_demo.py
"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: True
is comparison: False
```

References

[1] Python Software Foundation. "Built-in Types." https://docs.python.org/3/library/stdtypes.html (accessed August 2026).

[2] Python Software Foundation. "PEP 634 — Structural Pattern Matching: Specification." https://peps.python.org/pep-0634/ (accessed August 2026) — the match statement, introduced in Python 3.10.

[3] Python Software Foundation. "CPython's implementation of the -O flag and assert removal." https://docs.python.org/3/using/cmdline.html#cmdoption-O (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, an attribute, and a constructor each are, and why encapsulation matters. (2) Write a class from a plain-language specification: choose the right attributes, write an __init__ that sets all of them, and expose state only through methods or properties. (3) Write functions or 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: attributes hold an object's state, __init__ initializes that state exactly once at creation, and encapsulation (underscore-prefixed attributes, @property, public methods) is what keeps outside code from reaching in and corrupting that state directly. Lecture 3 then filled in Python's language fundamentals underneath all of that — numeric 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).

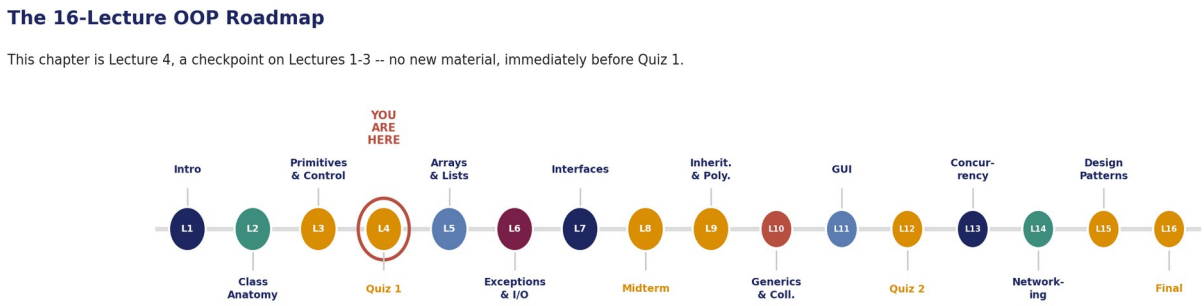


Figure 4.1 — This chapter sits at Lecture 4 in the sixteen-lecture OOP roadmap: a checkpoint, not a new topic, immediately before Quiz 1.

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 arithmetic, a plain function
2	Analog Clock Angle	L3 -- float arithmetic, modulo, normalization
3	Computer Account Class	L2 -- attributes, constructor, encapsulation
4	Complex Number Class	L2 -- attributes, methods returning new objects
5	Weight Conversion Class	L2 -- constructor, getters, a derived value
6	Advanced Fraction Class	L2/L3 -- __str__, 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 .py file 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 function compute(e1, d1, e2, d2) that returns 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 function 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. Python's tuple return and unpacking (return es, ds, ep, dp) is the idiomatic way to hand back four related numbers without needing a class yet — Problem 6 below is where a real Fraction class belongs.

Problem 2 *[Easy]*. Write a function angle_between(hours, 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, and the hour

hand creeps forward 0.5 degree per minute in addition to 30 degrees per hour. 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 with a float modulo. At 4:41 the raw angle is exactly 254.5 — think carefully about whether Python's built-in round() (banker's rounding) or int() (truncation toward zero) is the correct, defensible choice here, not just whichever one happens to produce the expected number for this one input.

4.3.2 Attributes, Constructors & Encapsulation

Problem 3 *[Easy]*. Define a class ComputerAccount, built from three strings: real_name, user_name, and password. Implement print_real_name(), print_user_name(), and print_password() (no arguments), plus change_password(new_password) (no return value).

> **Hint**
>
> This is the plainest possible constructor-plus-accessor-methods class: three attributes set in __init__, one method per attribute to print it, and one method that mutates a single attribute. Nothing here should require any new concept beyond Lecture 2.

Problem 4 *[Medium]*. Define a ComplexNumber class, storing a real part and an imaginary part, with a constructor and getters. Implement add, subtract, and multiply (each returning a new ComplexNumber), and a __str__ 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 ComplexNumber argument and must return a BRAND-NEW ComplexNumber, 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 a Weight class taking pounds in its constructor, with get_pounds() and get_kilograms() (using 1 pound = 0.45359237 kilograms).

> **Hint**
>
> Store only pounds as an attribute. get_kilograms() should compute the conversion on demand from that one attribute, rather than storing a second, redundant kilograms attribute that could fall out of sync if pounds were ever changed. The conversion constant itself belongs at module level, not as an instance attribute — it describes the conversion rule, not any one Weight.

4.3.3 Bringing It Together

Problem 6 *[Medium]*. Define a Fraction class with a constructor, get_numerator/get_denominator, a __str__ (e.g. "1/2"), and get_sum/get_product, each returning a new Fraction. For f1 = Fraction(1, 2) and f2 = Fraction(3, 7): str(f1) should return "1/2"; f2.get_product(f1) should print 3/14; f2.get_sum(f1) should print 13/14.

> **Hint**
>
> This is Problem 1's two formulas again, but this time get_sum and get_product must each return a fully-formed Fraction object, not a bare tuple — 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 take_damage(opponent_attack) (this side's health is reduced by opponent_attack - self.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 take_damage() apply its own defense and decide whether health has dropped below zero. Don't reach into the opponent's attributes 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 code that runs 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 code produce the exact expected result for every given test case?	50%
Class design & encapsulation	Are attributes underscore-prefixed, and is state reachable only through the constructor and methods?	25%
Code clarity	Is the code easy to read, idiomatic, and free of dead or duplicated code?	15%
Runs cleanly	Does it run under python3 with zero errors and no unhandled exceptions?	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 Python/FastAPI backend data model follows at far larger scale: underscore-prefixed attributes, a constructor that fully initializes them, and methods or properties 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 attributes, 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.

Code that runs without error is not the same as code 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 file run under python3 with zero unhandled exceptions?
- Is every attribute underscore-prefixed, with every value the caller needs exposed only through a method or @property?
- Does each class's __init__ set every attribute it uses — no attribute left undefined until some later method happens to set it?
- 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] Python Software Foundation. "Classes." The Python Tutorial. https://docs.python.org/3/tutorial/classes.html (accessed August 2026).

CHAPTER 5

Lists & the String-Building 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 Python's one built-in growable container, why Python has no separate fixed-size array to reach for, and the classic trap (and fix) of building a long string piece by piece.

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

(1) Create a list, and use append, remove, indexing, and len() to manage a growing collection. (2) Explain why Python has no separate fixed-size array type the way Java does, and what that means in practice. (3) Explain why repeated string concatenation with + inside a loop is quadratic, and fix it with the list-of-pieces-then-''.join() idiom. (4) State why Python has no public StringBuilder class, and what idiom takes its place. (5) Extend Pustaka with a Library class that manages a collection of Book objects, 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 Python's list, the one built-in container this course needs for 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, as Java's version of this same story (Figure 5.1).

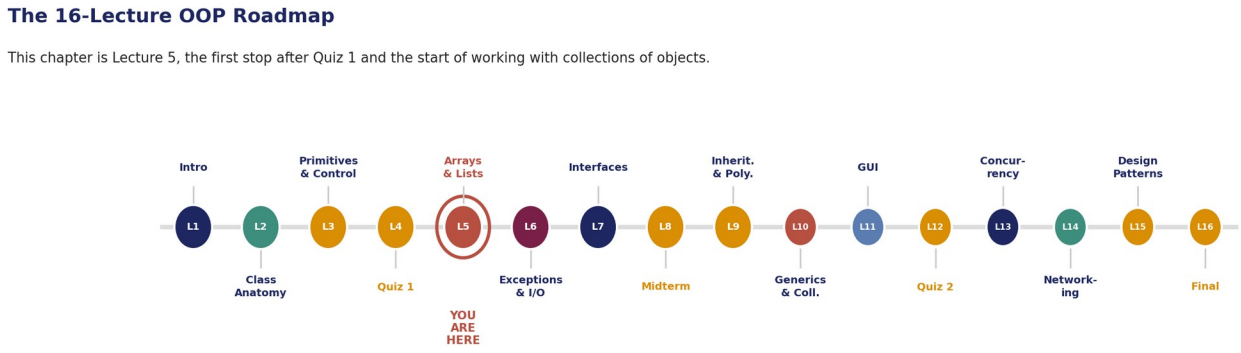


Figure 5.1 — The 16-lecture OOP roadmap. This chapter is Lecture 5, the first stop after Quiz 1 and the beginning of working with collections of objects.

5.2 Python's List: One Container, Always Growable

A Python list is created with square brackets -- shelf = [] starts empty -- and needs no length declared up front: shelf.append(book) grows it by one every time it is called, with no ceiling to ever hit. Unlike Java, Python has no separate fixed-size array type in ordinary use; the same list type serves every situation Java would split between array and ArrayList.

This is a genuine language design difference, not a missing feature: Java's array exists as a deliberately low-level, fixed-length primitive that ArrayList is built on top of internally; Python instead exposes only the dynamic version directly, because in practice almost every real collection -- Pustaka's shelf of books very much included -- needs to grow and shrink, and the fixed-size case is rare enough that Python does not give it a dedicated syntax (Figure 5.2).

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 Python's len() and append() read almost identically to ArrayList's size() and add().

5.3 Core List Operations

shelf.append(book) adds to the end; shelf[i] reads (or, unlike a Java array's fixed-length restriction, can also assign to) the item at index i; shelf.remove(book) removes by value (the first matching item); len(shelf) reports the current count. A plain for b in shelf: loop iterates a list directly -- no index variable needed unless the index itself matters, which is a small readability advantage Python's list has over indexed access in the common case.

> **One list, any type -- even mixed**
>
> Unlike Java's List<Book>, which the compiler enforces holds ONLY Book objects, a Python list can mix any types at once -- [1, "two", Book(...)] is perfectly legal syntax. In well-designed code you will almost always still keep one list to one conceptual type of item (exactly as Library._books does below, holding only Book objects) -- Python simply does not enforce that discipline for you the way Java's generics do; it is a convention you choose to follow, not a rule the language checks.

5.4 The String-Building Story

Strings, as Chapter 3 established, are immutable: every += on a str 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. This is exactly Java's StringBuilder story (Java Chapter 5, Section 5.4) -- the underlying cost trap is identical in both languages, because it comes from string immutability, not from any one language's specific design (Figure 5.3).

Figure 5.3 — The same report-building loop written the costly way (+= in a loop) and Java's fix, StringBuilder.append(). Python has no public builder class of its own -- see the callout below for Python's actual idiom.

> **Python's fix: a list of pieces, joined once**
>
> Java's fix is a mutable StringBuilder object; Python's idiomatic fix looks different but solves the exact same problem: collect each piece into a plain list (pieces.append(next_piece) -- append on a list is already O(1) amortized, no special builder needed), then call "".join(pieces) exactly once at the end. join() knows the total length in advance and allocates the final string once, so the whole operation is linear (O(n)), just like StringBuilder.append() followed by a single toString(). Python's io.StringIO offers a more Java-StringBuilder-like mutable buffer for very large or incremental text streams, but for the ordinary case this chapter covers, list-then-join is the idiom you will see in essentially all real Python code.

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: add_book and remove_book manage the collection, and generate_report builds a multi-line summary of the whole shelf using the list-then-join idiom rather than += .

Fase 5 — Implement (library.py, abridged)

```
class Library:
    def __init__(self) -> None:
        self._books: list[Book] = []

    def add_book(self, book: Book) -> None:
        self._books.append(book)

    def remove_book(self, isbn: str) -> bool:
        for i, b in enumerate(self._books):
            if b.isbn == isbn:
                del self._books[i]
                return True
        return False

    def generate_report(self) -> str:
        lines = [f"Pustaka Library Report ({len(self._books)} book(s))"]
        for b in self._books:
            status = " [on loan]" if b.is_on_loan else " [on shelf]"
            lines.append(f"- {b.title} by {b.author}{status}")
        return "\n".join(lines) + "\n"
```

Notice remove_book uses enumerate(self._books) to get both the index and the item in one loop -- Python's idiomatic way to need "the position of the matching item," here so del self._books[i] can remove it by position, mirroring Java's indexed for (int i = 0; ...) loop in Library.removeBook exactly, just with Python's own idiom for pairing an index with its value.

5.6 Running Your Program

Nothing changes about the process: python3 library_demo.py. library_demo.py imports both Book (from pustaka) and Library (from library) at the top of the file -- Python's module system, one file per class here, is the direct counterpart to Java needing each class in its own compiled .java file.

```
$ python3 library_demo.py
```

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 Python/FastAPI 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 describes Python's list handling natively, with no fixed-size alternative to even consider. 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 list-then-join discipline Section 5.4 introduces: collect the pieces first, join them once, never concatenate with += inside a loop.

5.8 Chapter Summary and Key Takeaways

- Python has exactly one built-in list type, always growable -- no separate fixed-size array the way Java has, because in practice almost every real collection needs to grow and shrink.
- A list's core operations are append, remove, indexing, and len() -- a plain for item in list: loop iterates it directly.
- Unlike Java's List<T>, a Python list is not restricted to one type by the language itself -- keeping one list to one conceptual type is a convention you choose to follow, not something Python enforces.
- str += inside a loop is quadratic (O(n^2)) because every += copies the whole string built so far; collecting pieces in a list and calling ''.join() once at the end is linear (O(n)).
- Python has no public StringBuilder-equivalent class; list-then-join is the idiomatic fix everywhere in real Python code (io.StringIO exists for very large streaming cases).
- Library gained add_book, remove_book, find_by_title, __len__, and generate_report 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. Java's Section 5.2 spends real space explaining an array's fixed-length limitation. Why does this chapter's Section 5.2 not need an equivalent warning about Python lists?
2. Rewrite Library.generate_report() using += inside the loop instead of the list-then-join idiom. For a library of 10,000 books, roughly how many times more total character-copying work would this version do compared to the join() version? (Hint: compare n vs. n^2 for n = 10,000.)
3. Python lists can legally mix types ([1, "two", Book(...)]). Give one concrete reason why Library._books should still never actually do this, even though Python would not stop it.
4. Add a method Library.count_on_loan() that returns how many books in the collection are currently on loan, using a plain for loop and Book's existing is_on_loan property.
5. In your own words, explain what io.StringIO is for, and why the ordinary list-then-join idiom is still preferred for a case as small as this chapter's library report.

Appendix 5.A — Execution Verification Log

The complete pustaka.py, library.py, and library_demo.py files (Code/Python/Chapter05/, provided alongside this book) were run on Python 3.13 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ python3 library_demo.py
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] Python Software Foundation. "Data Structures — More on Lists." The Python Tutorial. https://docs.python.org/3/tutorial/datastructures.html (accessed August 2026).

[2] Python Software Foundation. "str.join()." The Python Standard Library. https://docs.python.org/3/library/stdtypes.html#str.join (accessed August 2026).

[3] Python Software Foundation. "io — Core tools for working with streams (StringIO)." https://docs.python.org/3/library/io.html#io.StringIO (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 Python'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 Python's single exception model and how it differs from Java's checked/unchecked split. (2) Write a try-except-finally block, and state precisely when the finally block runs. (3) Define a custom exception by subclassing Exception, and raise and catch it correctly. (4) Read and write a plain text file using a with statement (context manager), and explain why it is preferred over manually calling close(). (5) Extend Pustaka with save/load persistence and a stricter, exception-raising removal method, 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. remove_book 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: Python'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).

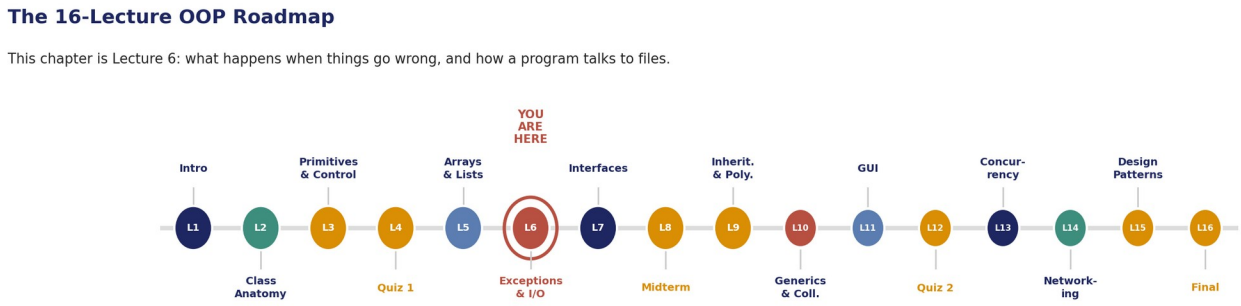


Figure 6.1 — The 16-lecture OOP roadmap. This chapter is Lecture 6: what happens when things go wrong, and how a program talks to files.

6.2 Python's Exception Model: One Category, No Compiler Enforcement

Every error condition in Python is represented by an object -- ultimately, every exception descends from BaseException. Ordinary code almost never catches BaseException directly, since its most direct subclasses include things like SystemExit and KeyboardInterrupt that represent "the program is being asked to stop," not an error to recover from. Exception is the branch that ordinary error handling actually works with. Unlike Java, Python draws no compiler-enforced line between "checked" and

"unchecked" exceptions: nothing forces a caller to catch anything, or to declare in a function's signature that it might raise. A caller finds out what a function can raise by reading its documentation (or its source) -- this is Python's EAFP style ("easier to ask forgiveness than permission"): try the operation, and be ready to catch what it might raise, rather than checking every precondition defensively beforehand (Figure 6.2).

Checked Exceptions (Java) vs. Python's One Model

Java forces the caller to acknowledge certain failures at compile time; Python trusts the caller and raises everything the same way.

Property	Java	Python
Exception categories	`Checked (must handle/declare) + Unchecked`	`One category -- no compiler-enforced distinction`
Declaring a risky method	`throws BookNotFoundException`	`nothing required -- caller finds out by reading docs`
Catching	`catch (BookNotFoundException e) { ... }`	`except BookNotFoundError as e:`
Custom exception	`extends Exception (checked) or RuntimeException`	`class X(Exception): ...`
Guaranteed cleanup	`try-with-resources / finally`	`with statement (context manager) / finally`

Neither model is "more correct" -- checked exceptions catch missing error handling at compile time, at the cost of ceremony; Python's EAFP style ("easier to ask forgiveness than permission") trusts the developer to read the documentation and catch what genuinely needs catching.

Figure 6.2 — Java's checked-exception model compared with Python's single, uniform model. This chapter's own BookNotFoundError raises exactly the same way any other Python exception does.

> **Trusting the caller is a real design trade-off, not a shortcut**
>
> Java's checked exceptions catch a forgotten error-handling path at compile time, at the cost of the throws-declaration ceremony seen throughout this course's Java track. Python's model has no such compile-time safety net -- BookNotFoundError (Section 6.4) could in principle be raised by a call site that never expects it, with the error only surfacing at runtime. Python's designers judged this an acceptable trade for a simpler, less ceremonious language; it means Python code carries a correspondingly higher responsibility to document what a function can raise, and to actually read that documentation.

6.3 try, except, else, and finally

A try block wraps code that might raise; one or more except clauses each handle one specific exception type; an optional else clause runs only if the try block completed with no exception at all; and an optional finally clause runs no matter what -- whether the try block completed normally, raised an exception that was caught, or raised one that was not caught and is about to propagate further up the call stack. (Java's try-catch-finally has no direct equivalent to else; it is a small extra Python offers, for code that should run only on success.) (Figure 6.3)

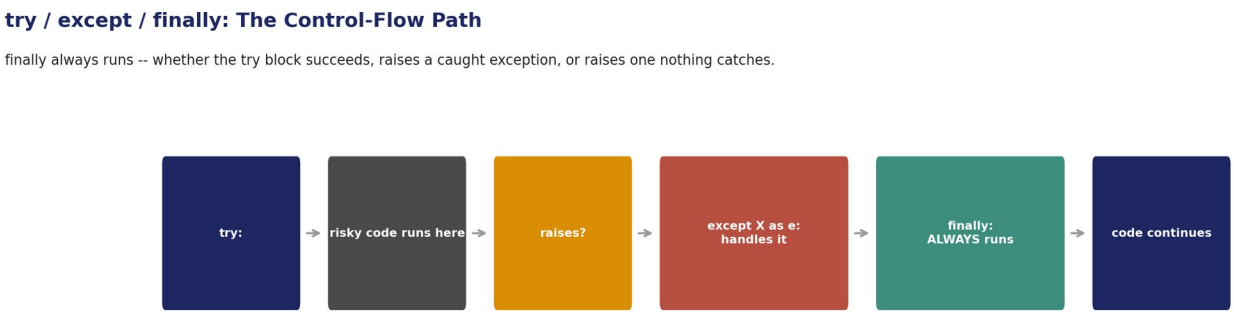


Figure 6.3 — The try/except/finally control-flow path (the same underlying structure as Java's try/catch/finally). finally's defining property is that it always executes.

A minimal try-except-finally

```
try:
    library.remove_book_or_raise(isbn)
except BookNotFoundError as e:
    print(f"Caught expected exception: {e}")
finally:
    print("This always prints, exception or not.")
```

A bare except: is almost always wrong

except: (with no exception type at all) catches literally everything, including KeyboardInterrupt and SystemExit, and an empty except block silently discards the failure entirely, which is far worse than letting the program crash with a clear traceback. Always catch the most SPECIFIC exception type your code can actually do something meaningful about -- except Exception: is only slightly better, and still broader than almost any real handler needs.

6.4 Custom Exceptions: BookNotFoundError

Defining a custom exception is ordinary inheritance: subclass Exception (never bare Exception itself for anything you plan to catch specifically, and never BaseException directly), call super().__init__(message) in __init__, and the new class is a fully legitimate exception type Python will let you raise and catch by name.

Phase 6 — Implement (book_not_found_error.py)

```
class BookNotFoundError(Exception):
    """Raised when a lookup or removal by ISBN finds no matching book."""

    def __init__(self, isbn: str) -> None:
        super().__init__(f"No book found with ISBN {isbn}")
        self.isbn = isbn
```

Library.remove_book_or_raise (Section 6.6) raises this exception when remove_book's ISBN lookup fails, giving callers a named, specific, catchable error condition instead of a generic ValueError or a silently-ignored False.

6.5 File I/O with the with Statement

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. Python's with statement, using any object that implements the context manager protocol (__enter__ and __exit__), guarantees the resource is closed when the with block exits, whether normally or via an exception -- no explicit finally block needed. open() returns exactly such an object, which is why with open(...) as f: is the idiomatic way to work with files in Python.

with statement vs. the old manual pattern

```
# Modern, idiomatic Python: the with statement -- always prefer this
with open(path, "r", encoding="utf-8") as f:
    line = f.readline()
    # f.close() is called automatically here, guaranteed

# Old manual pattern -- verbose and easy to get wrong
f = open(path, "r", encoding="utf-8")
try:
    line = f.readline()
finally:
    f.close()  # easy to forget, or to place incorrectly
```

OSError is Python's file-related exception

Every function that touches the filesystem can fail for reasons entirely outside your program's control -- a missing file, a full disk, a permissions error -- and Python raises OSError (or one of its more specific subclasses, like FileNotFoundError) for exactly these situations. Unlike Java's IOException, nothing forces Section 6.6's save_to_file and load_from_file to declare this in their signature -- Python's uniform exception model (Section 6.2) applies here exactly as everywhere else.

6.6 Extending Pustaka: Persistence and Stricter Removal

Library gains two related capabilities this chapter: remove_book_or_raise, a stricter sibling of remove_book (Chapter 5) that raises BookNotFoundError instead of quietly returning False; and save_to_file/load_from_file, simple text-file persistence built on the with statement.

Phase 6 — Implement (library.py, abridged)

```
def remove_book_or_raise(self, isbn: str) -> None:
    if not self.remove_book(isbn):
        raise BookNotFoundError(isbn)

def save_to_file(self, path: str) -> None:
    with open(path, "w", encoding="utf-8") as f:
        for b in self._books:
            f.write(f"{b.title}|{b.author}|{b.isbn}|{b.publication_year}\n")

def load_from_file(self, path: str) -> None:
    with open(path, "r", encoding="utf-8") as f:
        for line in f:
            title, author, isbn, year = line.rstrip("\n").split("|")
            self._books.append(Book(title, author, isbn, int(year)))
```

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

6.7 Running Your Program

Nothing changes about the process: python3 library_demo.py. library_demo.py imports Book (from pustaka), Library (from library), and BookNotFoundError (from book_not_found_error) at the top of the file.

```
$ python3 library_demo.py
```

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 Python/FastAPI 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 exception handling existing for. Every such call is wrapped in its own try/except with a specific, named exception type -- never a bare except:, 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 -- the Databases/FBP course covers that machinery directly), the underlying discipline is identical: never leave a resource open longer than it needs to be (the with statement, used everywhere Profitina's backend touches a file or a network connection), and never let an I/O failure pass unnoticed.

6.9 Chapter Summary and Key Takeaways

- Every Python exception descends from BaseException; ordinary error handling works with Exception, its main subclass branch.
- Unlike Java, Python has no compiler-enforced checked/unchecked distinction -- nothing forces a caller to catch anything or declare what a function might raise; this is Python's EAFP philosophy.
- try-except-finally: finally always runs, whether the try block succeeds, raises a caught exception, or raises one that propagates uncaught; else runs only when the try block succeeds with no exception at all.
- A custom exception is ordinary inheritance -- subclass Exception -- and BookNotFoundError (this chapter's) follows exactly that pattern.
- Always catch the most specific exception type possible; never leave an except block empty, and avoid a bare except: or overly broad except Exception:.
- The with statement automatically closes any context-manager resource (a file here) when the block exits, replacing the old, error-prone manual try-finally-close() pattern.
- Library gained remove_book_or_raise, save_to_file, and load_from_file 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. Explain in your own words why Python has no equivalent to Java's "checked exception" -- what does this mean for how a Python developer learns what a function can raise?
2. Rewrite the with-statement 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 the with statement makes impossible.
3. Why does Library.save_to_file let an OSError propagate to its caller instead of catching it internally and printing an error message?
4. Add a method Library.count_by_author(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 raise an exception or simply return 0.
5. In your own words, explain why a bare except: block is considered one of the worst mistakes a Python program can make, even though it runs without any visible error.

Appendix 6.A — Execution Verification Log

The complete book_not_found_error.py, pustaka.py, library.py, and library_demo.py files (Code/Python/Chapter06/, provided alongside this book) were run on Python 3.13 before this chapter

was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ python3 library_demo.py
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] Python Software Foundation. "Errors and Exceptions." The Python Tutorial. https://docs.python.org/3/tutorial/errors.html (accessed August 2026).

[2] Python Software Foundation. "with statement." The Python Language Reference. https://docs.python.org/3/reference/compound_stmts.html#with (accessed August 2026).

[3] Python Software Foundation. "Reading and Writing Files." The Python Tutorial. https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files (accessed August 2026).

[4] J. Bloch. Effective Java, 3rd ed. Addison-Wesley, 2017, Chapter 10 ("Exceptions") -- referenced here for the cross-language comparison in Section 6.2.

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 Python gives you for sharing structure and behavior across genuinely different classes -- abc.ABC, for types that share real state and code, and typing.Protocol, for capabilities that cut across any single hierarchy.

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

(1) Explain why an ABC subclass with an unimplemented @abstractmethod cannot be instantiated, and write one with both abstract and concrete methods. (2) Define a typing.Protocol and explain how a class satisfies it structurally, without ever inheriting from it. (3) State a working rule of thumb for choosing ABC versus Protocol for a given design problem. (4) Refactor an existing class to inherit from an abstract superclass and satisfy a Protocol, without breaking any of its existing behavior. (5) Write a loop that calls a method polymorphically across two unrelated concrete subtypes, 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: attributes and encapsulation via properties (Chapter 2), Python's type model and control flow (Chapter 3), lists and the StringBuilder/StringBuffer story (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)?

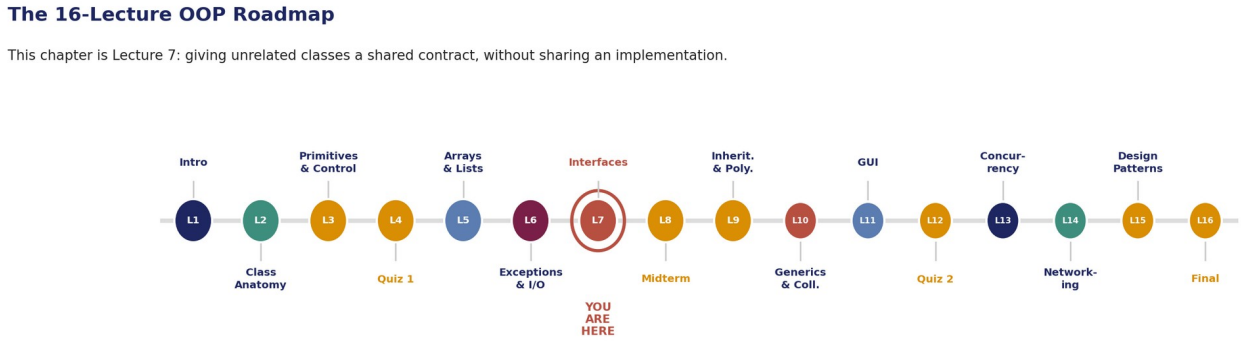


Figure 7.1 — The 16-lecture OOP roadmap. This chapter is Lecture 7: giving unrelated classes a shared contract, without sharing an implementation.

7.2 Python's Answer to Abstract Classes: ABC and @abstractmethod

Python has no abstract class keyword. The abc module's ABC base class, combined with the @abstractmethod decorator, is the standard-library tool that gets closest to Java's abstract classes: a class that inherits from ABC and declares one or more @abstractmethod methods cannot be instantiated directly, and Python raises a TypeError at construction time if a concrete subclass forgets to override one of them. This is one of the very few places Python enforces a contract at all --

everywhere else in this course, including exception handling (Chapter 6), Python trusts the caller instead.

A minimal ABC with an abstract method

```
from abc import ABC, abstractmethod

class LibraryItem(ABC):
    def __init__(self, title: str) -> None:
        self._title = title
        self._on_loan = False

    # Concrete method -- every subclass inherits this exact code.
    @property
    def is_on_loan(self) -> bool:
        return self._on_loan

    # Abstract method -- every subclass MUST supply its own version.
    @abstractmethod
    def calculate_fine(self, days_overdue: int) -> float:
        raise NotImplementedError
```

@abstractmethod is enforced at construction, not at class-definition time

Python happily lets you DEFINE a subclass that forgets to override an abstract method -- the TypeError only fires the moment something tries to instantiate it. A missing override can sit undetected in a large codebase until the exact line that tries to construct the incomplete subclass runs.

7.3 Python's Answer to Interfaces: Protocol and Structural Typing

typing.Protocol (PEP 544) takes a completely different, STRUCTURAL approach to "every conforming type must provide this." A class satisfies a Protocol simply by having methods with matching names and signatures -- no inheritance declaration of any kind is required, and nothing is ever written like class Book(Renewable). This is Python's duck-typing philosophy applied formally: "if it renews like a Renewable, it is one."

A minimal Protocol

```
from typing import Protocol, runtime_checkable

@runtime_checkable
class Renewable(Protocol):
    def renew(self) -> bool: ...
    def renewal_status_label(self) -> str: ...

# Book and DVD satisfy Renewable just by having these two methods --
# neither one ever writes `class Book(Renewable)` anywhere.
```

The @runtime_checkable decorator is what lets isinstance(some_item, Renewable) actually work at runtime, checking only that the right method names exist -- it does not check argument types or return types, which is a real limitation compared to Java's compiler-checked interfaces (Figure 7.2).

7.4 Comparing to Java, and Choosing Between ABC and Protocol

Four Ways to Say "Every Subtype Must Provide This"

Java draws a hard line between abstract classes and interfaces; Python's two tools are closer cousins than they look.

	Abstract class	Interface / Protocol
Java	abstract class + abstract method; single inheritance only	interface + default methods; a class may implement MANY
Python	ABC + @abstractmethod; enforced at construction time	typing.Protocol; structural -- no inheritance needed at all

Java's rule of thumb: reach for an abstract class when subtypes share real state and code (LibraryItem); reach for an interface when you are only describing a capability, unrelated to any single hierarchy (Renewable). Python's ABC mirrors the abstract-class case exactly; its Protocol is looser than any Java interface -- conformance is checked by shape, not by declaration.

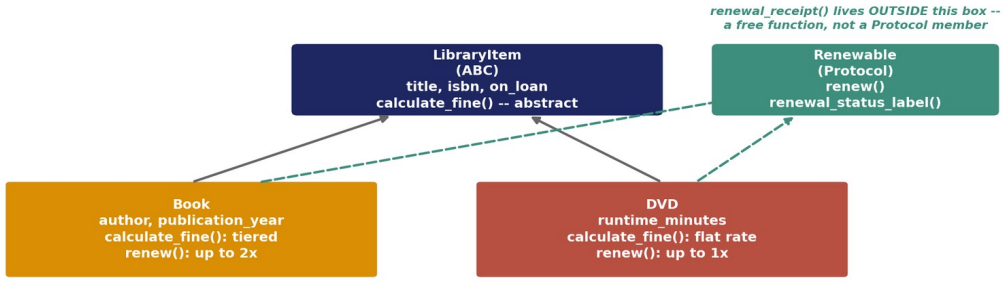
Figure 7.2 — Four ways to express "every subtype must provide this," across Java and Python.

Java draws a hard structural line: abstract class for single-inheritance shared state and code, interface for a capability a class may implement as many of as it needs, since Java 8 optionally carrying a default method with a full, inheritable implementation. Python's two tools map onto that same distinction, but Protocol is looser than any Java interface -- conformance is checked by shape, not by declaration, and Protocol itself cannot carry a shared implementation the way a Java default method can (Section 7.5 shows Python's usual workaround). A practical rule of thumb: reach for ABC when subtypes share real, non-trivial state and behavior that would otherwise be duplicated (LibraryItem's title, isbn, and on-loan state, plus its concrete borrow() logic); reach for Protocol when you are describing a capability that cuts across any single hierarchy (Renewable) (Figure 7.3).

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

Pustaka's Chapter 7 Shape: One Abstract Class, One Interface, Two Concrete Types

LibraryItem forces a shared shape; Renewable is a capability Book and DVD both opt into, independently of that shape.



Dashed arrows are "implements" (a capability); solid arrows are "extends" (an is-a relationship, one per class). LibraryItem cannot be instantiated directly -- only Book and DVD, its two concrete subclasses, can be.

Figure 7.3 — Pustaka's Chapter 7 shape: LibraryItem is an abstract base class; Renewable is a Protocol Book and DVD each satisfy structurally.

Book's title, isbn, and on-loan state, along with its borrow()/return_item() logic, move up into a new abstract base class, LibraryItem, which also declares calculate_fine() 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, publication_year, and times_renewed.

Phase 7 — Implement (library_item.py, abridged)

```
class LibraryItem(ABC):
    def __init__(self, title: str, isbn: str) -> None:
        self._title = title
        self._isbn = isbn
        self._on_loan = False

    def borrow(self) -> bool:
        if self._on_loan:
            return False
        self._on_loan = True
        return True

    @abstractmethod
    def calculate_fine(self, days_overdue: int) -> float:
        raise NotImplementedError

    def describe(self) -> str:
        status = "ON LOAN" if self._on_loan else "on the shelf"
        return f'"{self._title}" [ISBN {self._isbn}] - {status}'
```

DVD is a brand-new item type this chapter -- it shares nothing with Book except LibraryItem's state and Renewable's shape. 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.py, abridged)

```
_MAX_RENEWALS = 1
_DAILY_FINE = 0.75

class DVD(LibraryItem):
    def __init__(self, title: str, isbn: str, runtime_minutes: int) -> None:
        super().__init__(title, isbn)
        self._runtime_minutes = runtime_minutes
        self._times_renewed = 0

    def calculate_fine(self, days_overdue: int) -> float:
        return 0.0 if days_overdue <= 0 else days_overdue * _DAILY_FINE

    def renew(self) -> bool:
        if not self._on_loan or self._times_renewed >= _MAX_RENEWALS:
            return False
        self._times_renewed += 1
        return True

    def renewal_status_label(self) -> str:
        return (
            "No renewals left" if self._times_renewed >= _MAX_RENEWALS
            else "Renewable once"
        )
```

Java's default method vs. Python's free function

Java's Renewable interface can carry a default method (renewalReceipt) built on its own abstract methods, inherited for free by every implementing class. A Protocol has no equivalent -- it cannot carry a shared implementation, since a conforming class never actually inherits from it. Python's usual answer is a plain module-level function instead: renewal_receipt(item, attempt_succeeded), taking any Renewable-shaped object as its first argument.

7.6 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 calculate_fine(), renew(), and describe() are all called polymorphically -- the loop never once asks "is this a Book or a DVD?" Python dispatches each call to the correct override automatically, based on the object's actual runtime type (Figure 7.4).

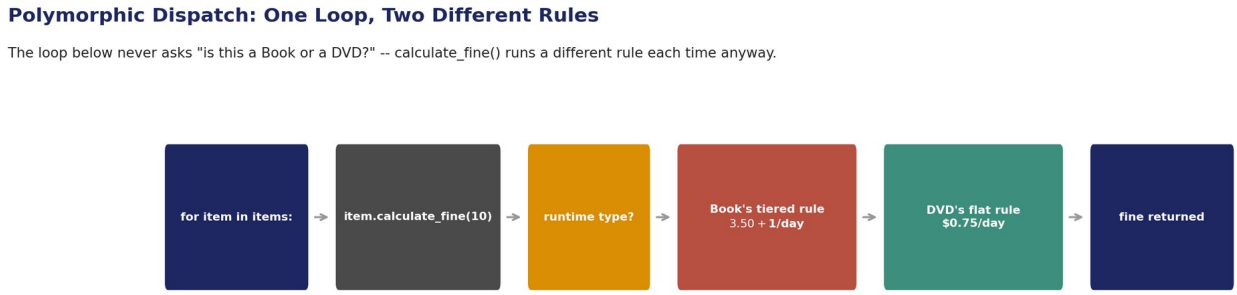


Figure 7.4 — Polymorphic dispatch: one loop body, two genuinely different calculate_fine() rules.

Phase 7 — Implement (library_demo.py, abridged)

```
items: list[LibraryItem] = [
    Book("Clean Code", "Robert C. Martin", "9780132350884", 2008),
    DVD("The Imitation Game", "9900000000001", 114),
]

for item in items:
    item.borrow()

for item in items:
    print(f"{item.describe()} -> fine: ${item.calculate_fine(10):.2f}")

for item in items:
    if isinstance(item, Renewable):
        ok = item.renew()
        print(f"{item.title}: {renewal_receipt(item, 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.

7.7 Running Your Program

Nothing changes about the process: python3 library_demo.py. library_demo.py imports LibraryItem, Book, DVD, Renewable, and renewal_receipt at the top of the file; no third-party packages are required beyond the standard library's abc and typing modules.

```
$ python3 library_demo.py
```

7.8 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 Python/FastAPI 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 shared Protocol (this chapter's exact pattern -- a structural contract, no shared implementation required) that every provider-specific connector satisfies, 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 with the right method shapes, not touching any of the code that already calls it.

7.9 Chapter Summary and Key Takeaways

- An ABC subclass with an unimplemented @abstractmethod cannot be instantiated -- Python enforces this at construction time, one of the few contracts it enforces at all.
- A typing.Protocol describes a capability structurally: a class satisfies it purely by having matching method names and signatures, with no inheritance declaration required.
- Rule of thumb: ABC for subtypes that share real state and code; Protocol for a capability that cuts across any single hierarchy.
- Java's abstract class and interface map onto Python's ABC and Protocol, but Protocol is looser than any Java interface, and cannot carry a shared implementation the way a Java default method can.
- LibraryItem (ABC) and Renewable (Protocol) 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 calculate_fine(), 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 base class and a Protocol are what let old code keep working without a single line of it changing.

7.10 Review Questions

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

1. Why does Python allow LibraryItem to be DEFINED with an unimplemented @abstractmethod, but refuse to let you instantiate it?
2. Add a third LibraryItem subtype, Magazine, that does NOT satisfy 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 two methods does it correctly omit?
3. Rewrite renewal_receipt so that it calls item.renew() itself instead of taking the outcome as a parameter. Explain the one concrete bug this introduces if a caller ever calls renewal_receipt() more than once.
4. In your own words, explain why @runtime_checkable Protocol's isinstance() check is weaker than Java's compile-time interface check -- what kind of mistake can pass Python's check but would be caught by Java's compiler?
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 library_item.py, renewable.py, book.py, dvd.py, and library_demo.py files (Code/Python/Chapter07/, provided alongside this book) were run on Python 3.13 before this chapter

was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ python3 library_demo.py
--- Overdue fines after 10 days (calculate_fine) ---
"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 (__str__, 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] Python Software Foundation. "abc — Abstract Base Classes." Python 3 documentation. https://docs.python.org/3/library/abc.html (accessed August 2026).

[2] Python Software Foundation. "typing — Protocol." Python 3 documentation. https://docs.python.org/3/library/typing.html#typing.Protocol (accessed August 2026).

[3] Oracle Corporation. "Abstract Methods and Classes" and "Default Methods." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/ (accessed August 2026, cross-language reference for Section 7.4).

[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 Protocols, 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 attribute 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 ABC where a concrete method is built on top of an abstract one, and a Protocol satisfied structurally, then dispatch over a list of that Protocol 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 — attributes, constructors, and encapsulation. Lecture 3 filled in Python's numeric types, control flow, strings, and basic debugging tools. Lecture 4 was Quiz 1, a first checkpoint on all of that. Lecture 5 introduced lists and the string-building story (concatenation versus join versus f-strings). 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 abc.ABC and typing.Protocol, letting Pustaka's Book and a brand-new DVD type share a contract without sharing an implementation (Figure 8.1).

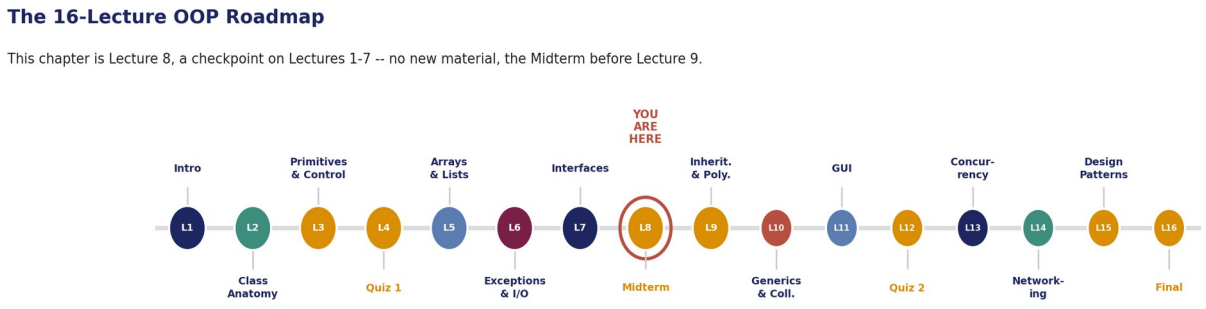


Figure 8.1 — This chapter sits at Lecture 8 in the sixteen-lecture OOP roadmap: a checkpoint, not a new topic, the Midterm before Lecture 9.

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 -- attributes, constructor, a derived value
2	Grade Converter	L3 -- control flow, boundary conditions
3	Roster Formatter	L5 -- string building, lists
4	Top-N Scores	L5 -- lists, order-preserving selection
5	Score Range Check	L6 -- a custom exception class
6	Score Log File	L6 -- file I/O, the with statement
7	Employee Pay	L7 -- ABC, a concrete method built on an abstract one
8	Discountable Items	L7 -- Protocol, structural typing, 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 .py file 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 Attributes, Constructors & Encapsulation Review

Problem 1 *[Easy]*. Define a class Temperature(celsius), storing only a Celsius value, with get_celsius(), get_fahrenheit() (F = C * 9/5 + 32), and get_kelvin() (K = C + 273.15).

Hint

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

8.3.2 Numeric Types & Control Flow Review

Problem 2 *[Easy]*. Write a function letter_grade(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, using elif, and return as soon as one matches. 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 Lists & String Building

Problem 3 *[Medium]*. Write a function format_roster(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 idiomatically rather than with repeated string concatenation in a loop.

> **Hint**
>
> Handle the 0/1/2-name cases as their own short-circuit branches first — they don't need any joining logic at all. For 3 or more, ", ".join(names[:-2]) handles everything except the last two, then append ", " + names[-2] + " and " + names[-1] — but watch the comma placement so 3 names don't get a stray leading ", ".

Problem 4 *[Medium]*. Write a function top_n(scores, 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 (sorted(scores, reverse=True)) to find the value at the cutoff position (the nth-largest value), then walk the ORIGINAL list in its original order with a list comprehension 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 exception class InvalidScoreError(Exception), and a function validate_score(score) that returns score unchanged if it is between 0 and 100 inclusive, and otherwise raises an InvalidScoreError whose message includes the invalid value.

> **Hint**
>
> InvalidScoreError needs no body beyond `pass` — inheriting from Exception, exactly like Chapter 6's BookNotFoundError. validate_score() only needs one if condition covering both out-of-range directions at once, then a single raise statement with an f-string message.

Problem 6 *[Medium]*. Write two functions: append_score_record(path, name, score), which appends one line "name,score" to the file at path, and read_score_records(path), which reads every line back and returns a list of (name, score) tuples, with score converted back to int.

> **Hint**
>
> Use with open(path, "a") as f: for the first function (append mode — do NOT use "w", which would erase the file's existing lines), and with open(path) as f: for the second, splitting each line on the comma and stripping the trailing newline before converting the score back to int — exactly the pattern Chapter 6's save_to_file/load_from_file used for Pustaka.

8.3.5 ABC & Protocol

Problem 7 *[Hard]*. Define an abstract class Employee(ABC) with a name attribute, an abstract method monthly_pay(self), and a CONCRETE method annual_pay(self) that returns self.monthly_pay() * 12. Then define two concrete subclasses: SalariedEmployee (a fixed monthly salary) and CommissionEmployee (a base monthly amount plus a commission rate multiplied by a monthly sales figure).

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

Problem 8 *[Hard]*. Define a @runtime_checkable Protocol named Discountable with a method discounted_price(self) -> float. Then define ClearanceItem(original_price) (discounted_price() = original_price * 0.5) and MemberItem(original_price, member_discount_rate) (discounted_price() = original_price * (1 - member_discount_rate)), neither one ever writing class ClearanceItem(Discountable) anywhere. Also write a free function describe_discount(item) that formats discounted_price() into a string. Finally, sum discounted_price() over a list[Discountable] containing a mix of both types.

> **Hint**
>
> Neither class declares any inheritance from Discountable at all — they satisfy it purely by having a matching discounted_price() method, exactly like Chapter 7's Book and DVD satisfying Renewable. describe_discount() must live OUTSIDE both classes as its own function, since a Protocol cannot carry a shared implementation the way a Java interface's default method can — the same reason Chapter 7's renewal_receipt() is a free function, not a Protocol member.

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 code that runs 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 function or class produce the exact expected result for every given test case, including boundary values?	45%
Class & Protocol design	Are attributes accessed only through methods, and is the ABC/Protocol 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 the with statement correctly?	15%
Code clarity & runs cleanly	Is the code easy to read, sensibly named, and free of dead code — and does it run 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's Python/FastAPI backend 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 ABC or Protocol 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), string building and lists (L5), custom exceptions and file I/O (L6), and ABC and Protocol (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.

Code that runs is not the same as code that is correct — and code that is correct for the round numbers is not the same as code 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 function and class run with zero errors on every given test case?
- Are attributes accessed only through methods, with no code outside the class reaching in directly?
- 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 top_n() 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 use the with statement rather than a manual open()/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] Python Software Foundation. "abc — Abstract Base Classes." Python 3 documentation. https://docs.python.org/3/library/abc.html (accessed August 2026).

[3] Python Software Foundation. "typing — Protocol." Python 3 documentation. https://docs.python.org/3/library/typing.html#typing.Protocol (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 an object's static (declared/annotated) type and its runtime type, and why method calls dispatch on the latter. (2) Write a subclass that extends an already-concrete class (not just an ABC), and decide correctly whether an override should call super() or replace the inherited behavior entirely. (3) Override __eq__ and __hash__ together, correctly, and explain why Python treats overriding one without the other as a bug serious enough to make a class unhashable. (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 ABC holding title, isbn, and on_loan plus an abstract calculate_fine() every concrete subtype must supply; Renewable, a Protocol 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.

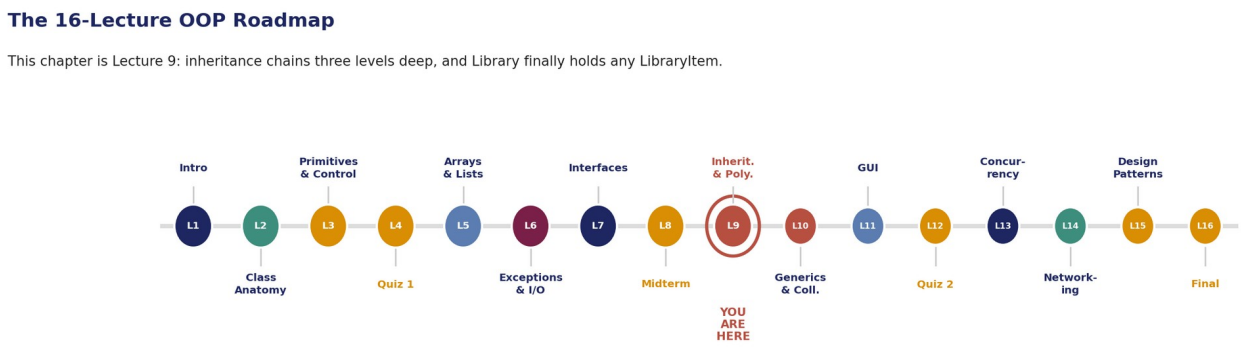


Figure 9.1 — The 16-lecture OOP roadmap. This chapter is Lecture 9: inheritance chains three levels deep, and Library finally holds any LibraryItem.

9.2 Inheritance, Formally: Subclassing, super(), and the Override Contract

A class that inherits from 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's

__init__ typically calls super().__init__(...) as its first step, forwarding arguments up the chain -- Python does not require this the way Java automatically inserts an implicit no-argument super() call whenever a constructor omits one, but skipping it usually means the parent's own attributes never get set up. Inside an overriding method, super().method_name(...) 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)

```
def borrow(self) -> bool:
    if not super().borrow():
        return False
    self._times_renewed = 0
    return True
```

Overriding does not require calling super()

Nothing about method overriding in Python 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 attribute and method Book has, including ones it will never use (_times_renewed, renewal_status_label()'s tiered wording), and overrides exactly four methods to enforce one rule: a reference item never leaves the building (Figure 9.2).

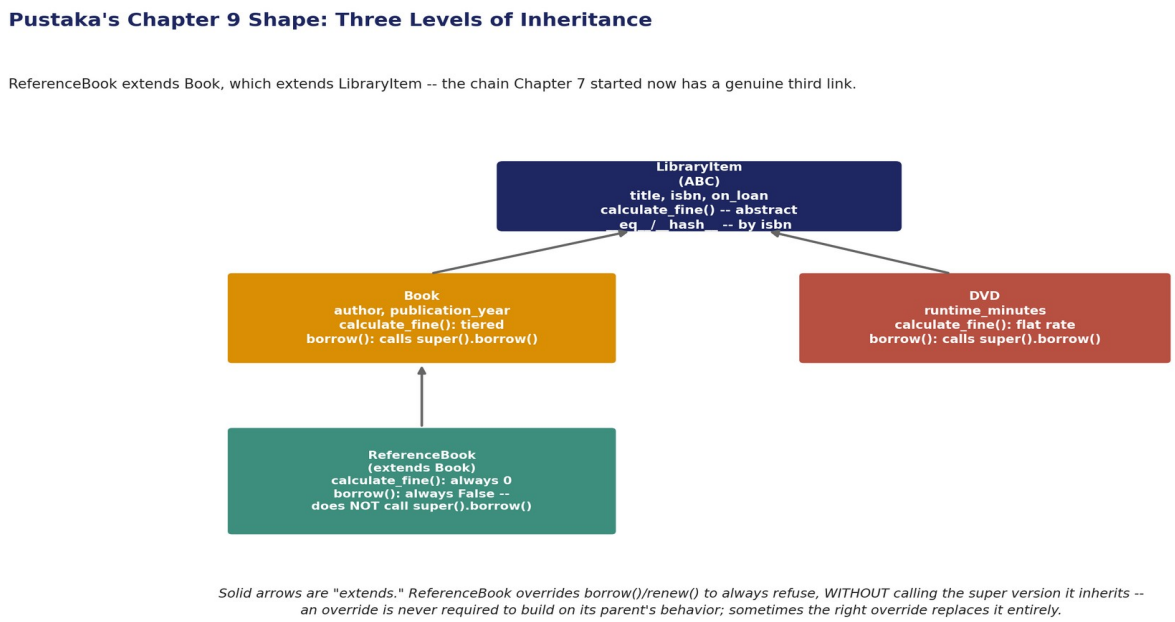


Figure 9.2 — Pustaka's Chapter 9 shape: LibraryItem → Book → ReferenceBook, a genuine third link in the chain Chapter 7 started.

reference_book.py (abridged -- all four overrides shown)

```
class ReferenceBook(Book):
    def __init__(self, title, author, isbn, publication_year=0):
        super().__init__(title, author, isbn, publication_year)

    # Deliberately does NOT call super().borrow() -- replaces it entirely.
    def borrow(self) -> bool:
        return False

    def renew(self) -> bool:
        return False

    def renewal_status_label(self) -> str:
        return "Reference only -- cannot be renewed"

    def calculate_fine(self, days_overdue: int) -> float:
        return 0.0
```

Single-underscore attributes reach across every level, not just one

self._title and self._isbn are set by LibraryItem.__init__, two levels above ReferenceBook -- Python's "protected by convention" attributes are inherited transitively like any other attribute, so ReferenceBook.__str__ can read self.title and self.author (via Book's own properties) directly, without LibraryItem granting that access a second time.

9.4 Polymorphism, Formally: Static Type vs. Runtime Type

A parameter annotated item: LibraryItem carries that annotation only for readability and for tools like mypy -- Python itself never checks it at runtime. What matters when item.calculate_fine(10) executes is item's ACTUAL runtime type: Python looks up calculate_fine on whatever class the object was actually built from and calls that version, every time. This is dynamic dispatch, and it is the entire mechanism behind every polymorphic loop this course has written since Chapter 7 -- arguably more central to Python than to Java, since Python never checks the annotation at all.

Passing a Book or a DVD anywhere a LibraryItem is expected (upcasting, in Java's terms) needs no special syntax in Python at all -- there is no cast to write. Narrowing the other direction (treating a LibraryItem you suspect is really a DVD as a DVD, to reach a DVD-only attribute like runtime_minutes) uses isinstance(), which checks first and only proceeds if the guess was correct -- Python has no equivalent to Java's checked downcast that can raise at the point of the cast itself; get the type wrong and an AttributeError surfaces instead, at the point runtime_minutes is actually accessed.

Passing a subtype needs no syntax; narrowing uses isinstance()

```
item: LibraryItem = DVD("The Imitation Game", "9900000000001", 114)  # no cast
needed

if isinstance(item, DVD):        # narrow, checked first
    print(f"{item.runtime_minutes} minutes")
```

9.5 Every Object Secretly Inherits from object: __eq__ and __hash__

Every class in Python -- whether it says so or not -- ultimately inherits from the built-in object, and inherits object's default __eq__ and __hash__: identity-based comparisons, equivalent to is and to id(), respectively. Two separately-constructed Book objects built from the exact same title, author, and ISBN are NOT == 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.__eq__ and __hash__ (abridged)

```
def __eq__(self, other: object) -> bool:
    if self is other:
        return True
    if not isinstance(other, LibraryItem):
        return NotImplemented
    return self._isbn == other._isbn

def __hash__(self) -> int:
    return hash(self._isbn)
```

Python will not let you override only one of the two, silently

The contract is the same as Java's -- if a == b is True, hash(a) must equal hash(b) -- but Python enforces the missing-half case more aggressively than Java does: defining __eq__ WITHOUT also defining __hash__ makes the class unhashable (Python sets __hash__ to None automatically), since it can no longer assume the inherited hash still agrees with the new equality. (Java's compiler stays silent about the same mistake -- see Figure 9.3.)

9.6 Generalizing Library: One Collection, Any LibraryItem

Library's private attribute changes from list[Book] to list[LibraryItem] this chapter, and every method built on top of it -- add_item (was add_book), remove_item, remove_item_or_raise (was remove_book_or_raise, and now raises the generalized ItemNotFoundError below), find_by_title,

generate_report -- 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 save_to_file/load_from_file need an explicit type tag as the first field on each line.

library.py -- generalized add_item and generate_report (abridged)

```
class Library:
    def __init__(self) -> None:
        self._items: list[LibraryItem] = []

    def add_item(self, item: LibraryItem) -> None:
        self._items.append(item)

    # Polymorphic on purpose: never checks which concrete type it holds.
    def generate_report(self) -> str:
        lines = [f"Pustaka Library Report ({len(self._items)} item(s))"]
        for item in self._items:
            lines.append(f"- {item.describe()}")
        return "\n".join(lines) + "\n"
```

library.py -- the type-tagged file format (abridged)

```
def _to_line(self, item: LibraryItem) -> str:
    if isinstance(item, ReferenceBook):
        return f"REFBOOK|{item.title}|{item.author}|{item.isbn}"
    elif isinstance(item, Book):
        return f"BOOK|{item.title}|{item.author}|{item.isbn}"
    elif isinstance(item, DVD):
        return f"DVD|{item.title}|{item.isbn}|{item.runtime_minutes}"
    raise TypeError(f"Unknown LibraryItem subtype: {type(item).__name__}")
```

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 BookNotFoundError is renamed ItemNotFoundError this chapter, for the same reason Library's own methods were renamed -- an exception's name should describe what it actually reports on, and "BookNotFoundError" 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 __eq__/__hash__ (a second Book object sharing an ISBN with the first), isinstance() narrowing for the one operation that is genuinely DVD-only, and a full save/load round trip through the type-tagged file format.

library_demo.py (abridged)

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

book = library.find_by_title("Clean Code")
dvd = library.find_by_title("The Imitation Game")
ref_book = library.find_by_title("Encyclopedia of Computer Science")

copy_of_same_book = Book("Clean Code (2nd copy)", "Robert C. Martin",
"9780132350884", 2008)
print(book == copy_of_same_book)  # True -- same ISBN, different object

for item in (book, dvd, ref_book):
    if isinstance(item, DVD):
        print(f"{item.runtime_minutes} minutes")
```

9.8 Compiling and Running Your Program

Python has no separate compile step -- python3 library_demo.py runs library.py, book.py, dvd.py, reference_book.py, item_not_found_error.py, and library_item.py the moment they are imported. No new packages beyond Chapter 7's abc and typing are required.

```
$ python3 library_demo.py
```

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 __eq__/__hash__ 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 __hash__ 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 actual runtime type, never a type annotation -- this is the entire mechanism behind every polymorphic loop since Chapter 7.
- Passing a subtype where a supertype is expected needs no special syntax in Python; narrowing back down uses isinstance(), 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 single-underscore attribute set at the top is still reachable at the bottom.
- Every class implicitly inherits from object and its identity-based __eq__/__hash__; overriding them together (never just one) switches a class over to value-based equality, built from the same attribute(s) in both methods -- Python disables hashing automatically if only __eq__ is overridden.
- 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.__eq__ 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 estimated_value attribute. Which of ReferenceBook's methods, if any, does RareBook need to override to change behavior, versus simply inherit unchanged?
4. Library._to_line() checks isinstance(item, ReferenceBook) before isinstance(item, 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 the AttributeError that surfaces when isinstance() narrowing is skipped and the guess turns out wrong, and the instanceof-style check used throughout this chapter -- which one can actually fail at runtime, and which one cannot?

Appendix 9.A — Execution Verification Log

The complete library_item.py, book.py, dvd.py, reference_book.py, renewable.py, item_not_found_error.py, library.py, and library_demo.py files (Code/Python/Chapter09/, provided

alongside this book) were executed on Python 3.13 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ python3 library_demo.py
--- 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())

--- __eq__/__hash__: value equality, not identity ---
book is copy_of_same_book        -> False  (different objects in memory)
book == copy_of_same_book        -> True  (same ISBN -> same real-world item)
hash(book) == hash(copy)         -> True

--- isinstance() narrowing: 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

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

References

[1] Python Software Foundation. "Data model — object.__eq__, object.__hash__." Python 3 documentation. https://docs.python.org/3/reference/datamodel.html#object.__hash__ (accessed August 2026).

[2] Oracle Corporation. "Overriding and Hiding Methods" and "Polymorphism." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/override.html and https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html (accessed August 2026) — cited for the Java-track comparison in Figure 9.3.

[3] 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") — the same discipline this chapter applies to Python's __eq__/__hash__.

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 Python's dict index alongside Library's list, a reusable generic Pair class, a generic bounded-type-parameter function, an __lt__-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 in Python's PEP 695 syntax, and why -- unlike Java -- nothing about them is ever checked by the interpreter itself, at any stage. (2) Read and write a generic function with a bounded type parameter (def total_fines[T: LibraryItem](...)), and explain what the bound T: LibraryItem restricts and what it does not. (3) Add a second dict-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 __lt__ (with @total_ordering) so sorted() produces a natural ordering with no key= argument, and explain what that ordering is being derived from. (5) Explain why Python's built-in set makes no ordering guarantee at all, and why a demo driver that prints one needs to wrap it in sorted() to keep a verification log reproducible.

10.1 Recap: Pustaka So Far

Chapter 9 generalized Library's attribute 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 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 Python's two other built-in collection types this course has not yet needed: dict and set.

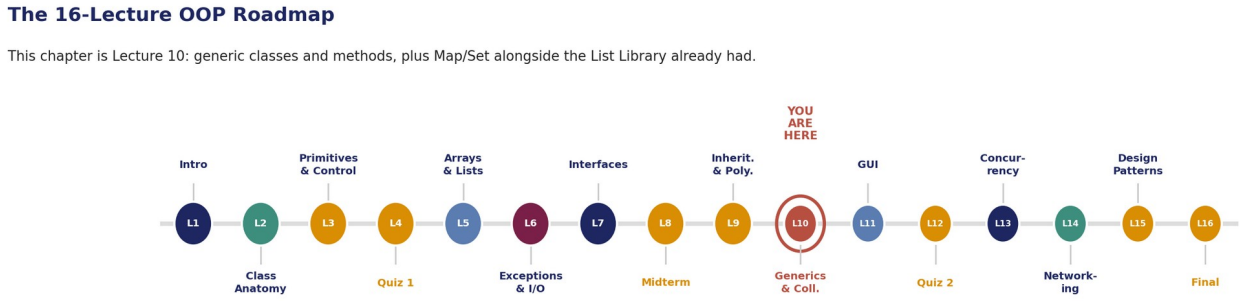


Figure 10.1 — The 16-lecture OOP roadmap. This chapter is Lecture 10: generic classes, a generic bounded function, and a dict/set-based second index alongside the list Library already had.

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, using Python's modern (3.12+) type-parameter syntax: A and B are type parameters, decided by the caller at the point of use (Pair[LibraryItem, float], Pair[str, int], ...) -- the class itself is written exactly once, just like Java's Pair<A, B> (Figure 10.1).

pair.py (complete)

```
class Pair[A, B]:
    def __init__(self, first: A, second: B) -> None:
        self._first = first
        self._second = second

    @property
    def first(self) -> A:
        return self._first

    @property
    def second(self) -> B:
        return self._second

    def __repr__(self) -> str:
        return f"({self._first}, {self._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).

Not "erasure" -- never enforced in the first place

Java's compiler checks every use of Pair<LibraryItem, Double> for consistency and only THEN discards the generic information (type erasure). Python has no equivalent checking step to discard: CPython never reads A and B as rules at all, at any stage -- they exist purely as metadata for an external tool like mypy or pyright to check, if one is run separately. The practical result is similar (neither language's RUNNING PROGRAM enforces the type parameters), but "erased after checking" and "never checked to begin with" are not the same claim, and Figure 10.2 keeps them visually distinct for exactly that reason.

10.3 A Generic Function with a Bounded Type Parameter: total_fines[T: LibraryItem]

total_fines() needs to add up calculate_fine() 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. Python 3.12's type-parameter syntax lets the function declare exactly that: def total_fines[T: LibraryItem](items: list[T], ...) reads as "T can be LibraryItem or any subtype of it," the bound after the colon, and the function body can then accept a list[Book] or a list[LibraryItem] equally, with no wildcard syntax needed at all.

library_utils.py (complete)

```
from library_item import LibraryItem


def total_fines[T: LibraryItem](items: list[T], days_overdue: int) -> float:
    return sum(item.calculate_fine(days_overdue) for item in items)
```

Python needs no wildcard here -- Java's List<? extends LibraryItem> exists to route around invariant, erased generics

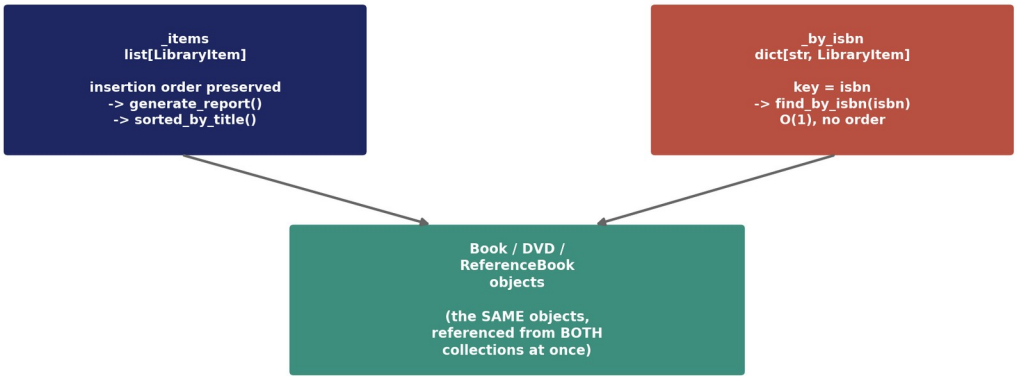
This IS a true generic function -- it declares its own type parameter, T, bound by LibraryItem. The Java track's equivalent (LibraryUtils.totalFines(), Section 10.3 of the Java-track book) instead uses a bounded WILDCARD, List<? extends LibraryItem>, not a type parameter -- because Java's List<Book> is not a List<LibraryItem> (generics are invariant there) and the wildcard exists specifically to work around that restriction. Python's list[Book] passed where list[LibraryItem] is annotated raises no such conflict in the first place, since nothing enforces the annotation at runtime -- so Python reaches directly for the more natural generic-function form instead of needing a wildcard-style escape hatch.

10.4 One Collection, Two Indexes: _by_isbn Alongside _items

Library's _items attribute (a list[LibraryItem], unchanged since Chapter 9) is joined this chapter by a second attribute, _by_isbn, a dict[str, LibraryItem] keyed on ISBN -- Python's built-in Map equivalent, needing no import at all, unlike Java's java.util.HashMap. add_item and remove_item now update both collections together, every time, so the two never drift out of sync. find_by_isbn(isbn) then answers in O(1) what find_by_title() still answers in O(n): both search the exact same underlying LibraryItem objects, just through two different paths (Figure 10.3).

Pustaka's Chapter 10 Shape: One Collection, Two Indexes

Library keeps a List AND a Map of the SAME LibraryItem objects -- not a copy, two access paths into one set of objects.



add_item()/addItem() and remove_item()/removeItem() update BOTH collections together, every time -- if they ever drifted out of sync, find_by_isbn()/findByIsbn() could return a stale or missing result even though the object is still sitting right there in the list.

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

library.py -- the _by_isbn index (abridged)

```
class Library:
    def __init__(self) -> None:
        self._items: list[LibraryItem] = []
        self._by_isbn: dict[str, LibraryItem] = {}

    def add_item(self, item: LibraryItem) -> None:
        self._items.append(item)
        self._by_isbn[item.isbn] = item

    def remove_item(self, isbn: str) -> bool:
        for i, item in enumerate(self._items):
            if item.isbn == isbn:
                del self._items[i]
                del self._by_isbn[isbn]
                return True
        return False

    def find_by_isbn(self, isbn: str) -> LibraryItem | None:
        return self._by_isbn.get(isbn)
```

Why keep the list at all, once there is a dict?

A dict alone would answer find_by_isbn() just as well, and (as of CPython 3.7) even preserves insertion order -- but relying on that as an ordering GUARANTEE would be relying on an implementation detail of iteration, not on what generate_report() and this chapter's sorted_by_title() actually need, which is a stable starting point they then sort or format explicitly. Library keeps both structures because they serve two genuinely different jobs: the list is the explicit source of truth for order, the dict is a second index built purely for O(1) lookup speed.

10.5 Natural Ordering: @total_ordering and __lt__

sorted(items) (or items.sort()) needs to know how to put two LibraryItems in order, and it gets that either from an explicit key= argument or from the element type itself supplying __lt__ -- 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 key= just to sort a list. @total_ordering (from functools) fills in __le__, __gt__, and __ge__ automatically once __eq__ (Chapter 9) and __lt__ (this chapter) both exist.

library_item.py -- __lt__() (new this chapter)

```
from functools import total_ordering

@total_ordering
class LibraryItem(ABC):
    # ... title, isbn, __eq__/__hash__ unchanged from Chapter 9 ...

    def __lt__(self, other: "LibraryItem") -> bool:
        if not isinstance(other, LibraryItem):
            return NotImplemented
        return self._title.lower() < other._title.lower()
```

library.py -- sorted_by_title()

```
def sorted_by_title(self) -> list[LibraryItem]:
    return sorted(self._items)
```

Natural ordering vs. an explicit key=

__lt__ 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 sorted(items, key=lambda i: i.calculate_fine(10)) overrides the natural order for that one call, without changing __lt__ or affecting any other caller.

10.6 A set of Distinct Types -- and Why the Demo Sorts It

distinct_types() 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 Python's built-in set (no import needed, unlike Java's HashSet/LinkedHashSet) makes NO promise at all about the order its elements come back in when iterated.

library.py -- distinct_types()

```
def distinct_types(self) -> set[str]:
    return {type(item).__name__ for item in self._items}
```

A plain Python set is not just unordered -- it can differ between runs of the SAME program

The Java track solves this exact reproducibility problem by choosing LinkedHashSet, a Set implementation that deliberately preserves first-seen order. Python's built-in set has no ordered variant to reach for at all, and worse: because CPython randomizes string hashing per process by default (PYTHONHASHSEED), the very same distinct_types() call can print its elements in a different order on two separate runs of the identical program -- confirmed by running the same one-liner three times and observing two different orderings. Appendix 10.A's log needs to be an honest, REPRODUCIBLE transcript, so library_demo.py wraps only its two print statements in sorted(...) for display -- distinct_types() itself still returns a genuine, unordered set[str]; nothing about its real return type or contract changes.

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) find_by_isbn() lookup, sorted_by_title() feeding directly into the bounded generic total_fines(), distinct_types() (Section 10.6's excerpt already shows this one, plus the sorted() display wrapper it needs), and finally two Pair 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/Python/Chapter10/library_demo.py.

library_demo.py (abridged)

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

found_by_isbn = library.find_by_isbn("9780321573513")
print(found_by_isbn.title)

by_title = library.sorted_by_title()
all_fines = total_fines(by_title, 10)
print(f"${all_fines:.2f}")

book = library.find_by_title("Clean Code")
most_expensive = Pair(book, book.calculate_fine(10))
print(most_expensive)
title_and_year = Pair("Clean Code", 2008)
print(title_and_year)
```

10.8 Compiling and Running Your Program

Python has no separate compile step, but this chapter's code needs a newer interpreter than any prior chapter: pair.py's class Pair[A, B]: and library_utils.py's def total_fines[T: LibraryItem](...) both use PEP 695 type-parameter syntax, introduced in Python 3.12. This is the first chapter where that requirement is not just recommended but load-bearing -- running library_demo.py on Python 3.11

raises a SyntaxError on the class Pair[A, B]: line itself, before any of this chapter's logic even runs. This course's environment defaults to Python 3.11, so Chapter 10 onward must be run explicitly with python3.12 or python3.13.

```
$ python3.13 library_demo.py
```

python3 alone is not enough from this chapter on

Every prior chapter's code ran fine under this environment's default python3 (3.11.15). pair.py and library_utils.py are the first files in this course that genuinely require 3.12+ syntactically -- python3 library_demo.py fails immediately with SyntaxError: invalid syntax on the very first PEP 695 class/function definition it imports, not on anything Chapter 10 actually does at runtime. Check your own interpreter with python3 --version, and if it reports 3.11 or earlier, use python3.12 or python3.13 explicitly instead, as this chapter's log below does.

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 dict keyed by a canonical external identifier for O(1) deduplication lookups when a new mention arrives -- precisely this chapter's _items/_by_isbn 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-attribute class for every score type it ever adds.

10.10 Chapter Summary and Key Takeaways

- Python's generic type parameters (Pair[A, B], PEP 695) are never checked by the interpreter itself, at any stage -- unlike Java, where the compiler checks once and then erases; only an external tool like mypy enforces them, if one is run at all.
- A generic function with a bounded type parameter (def total_fines[T: LibraryItem](...)) needs no wildcard-style escape hatch the way Java's equivalent does, since Python's type hints carry no runtime invariance restriction to route around.
- Library now keeps a dict[str, LibraryItem] (_by_isbn) alongside its list[LibraryItem] (_items), updated together on every insert and removal, trading the dict's unreliable ordering guarantee for O(1) ISBN lookup.
- Defining __lt__ (with @total_ordering) gives a class a natural ordering that sorted() uses with no key= argument; an explicit key= can still override that default for any one call, without touching __lt__.
- A plain Python set makes no ordering guarantee and can vary between runs of the identical program (string-hash randomization) -- distinct_types() keeps its honest set[str] return type,

and only the demo's display code wraps the result in sorted() to keep the verification log reproducible.

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 dict 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. total_fines() is written as def total_fines[T: LibraryItem](items: list[T], ...). Explain what would change, if anything, about which calls compile-check successfully under mypy if the bound were removed entirely (def total_fines[T](...)).
2. Suppose Library.remove_item(isbn) forgot to del self._by_isbn[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. Run python3.13 -c "print({'Book', 'DVD', 'ReferenceBook'})" three separate times. Explain, in your own words, why the three printed orderings are not guaranteed to be the same, and why that is a property of the string hashing, not of set itself being broken.
4. library_item.py's __lt__ orders by title, case-insensitively. Write an alternative sort call using sorted(items, key=...) that orders by calculate_fine(10) descending instead, without changing __lt__ at all.
5. Explain, in your own words, why pair.py's __repr__ uses str()-style formatting (f"({self._first}, {self._second})") rather than repr() on each element, and what would have printed instead for a Book if it had used !r formatting.

Appendix 10.A — Execution Verification Log

The complete library_item.py, renewable.py, book.py, dvd.py, reference_book.py, item_not_found_error.py, pair.py, library_utils.py, library.py, and library_demo.py files (Code/Python/Chapter10/, provided alongside this book) were executed on Python 3.13 before this chapter was written, to confirm the code is correct, not merely "looks correct." Output is reproduced verbatim below.

```
$ python3.13 library_demo.py
--- 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())

--- __eq__/__hash__: value equality, not identity (Chapter 9) ---
book == copy_of_same_book        -> True  (same ISBN -> same real-world item)

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

--- NEW Chapter 10: sorted_by_title(), via LibraryItem's __lt__ ---
  Algorithms
  Clean Code
  Encyclopedia of Computer Science
  The Imitation Game

--- NEW Chapter 10: distinct_types(), a set[str] (no duplicates) ---
distinct_types() -> ['Book', 'DVD', 'ReferenceBook']

--- NEW Chapter 10: total_fines(), a generic function with a bounded type
parameter ---
total_fines(all items, 10 days overdue) -> $20.50

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

--- remove_item_or_raise 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] Python Software Foundation. "Generic classes" and PEP 695 — "Type Parameter Syntax." Python 3 documentation and Python Enhancement Proposals. https://docs.python.org/3/reference/compound_stmts.html#generic-classes and https://peps.python.org/pep-0695/ (accessed August 2026).

[2] Python Software Foundation. "functools.total_ordering." Python 3 documentation. https://docs.python.org/3/library/functools.html#functools.total_ordering (accessed August 2026).

[3] Oracle Corporation. "Generics (Updated)" and "Bounded Types." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/generics/index.html and https://docs.oracle.com/javase/tutorial/java/generics/bounded.html (accessed August 2026) — cited for the Java-track comparison in Figure 10.2.

[4] Python Software Foundation. "object.__hash__" and "PYTHONHASHSEED." Python 3 documentation. https://docs.python.org/3/reference/datamodel.html#object.__hash__ and https://docs.python.org/3/using/cmdline.html#envvar-PYTHONHASHSEED (accessed August 2026) — cited for the non-deterministic set ordering discussed in Section 10.6.

CHAPTER 11

GUI Programming: Events & MVC

Every prior chapter's Pustaka has been a console program: a driver script 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 Python's built-in tkinter library 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 Tkinter event handler bound with a widget's command= argument, and explain when Tkinter actually calls it. (3) Build a Tkinter View class whose only job is constructing and exposing widgets, 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 invoking 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 dict[str, LibraryItem] keyed on ISBN, alongside its list[LibraryItem]), a reusable generic Pair[A, B] class (PEP 695), a true generic utility function, and a natural ordering via __lt__ -- but every chapter through Chapter 10 still interacts with that Library through nothing more than a sequence of print() calls chosen once, at the time the script was written. Nothing in library_demo.py 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.

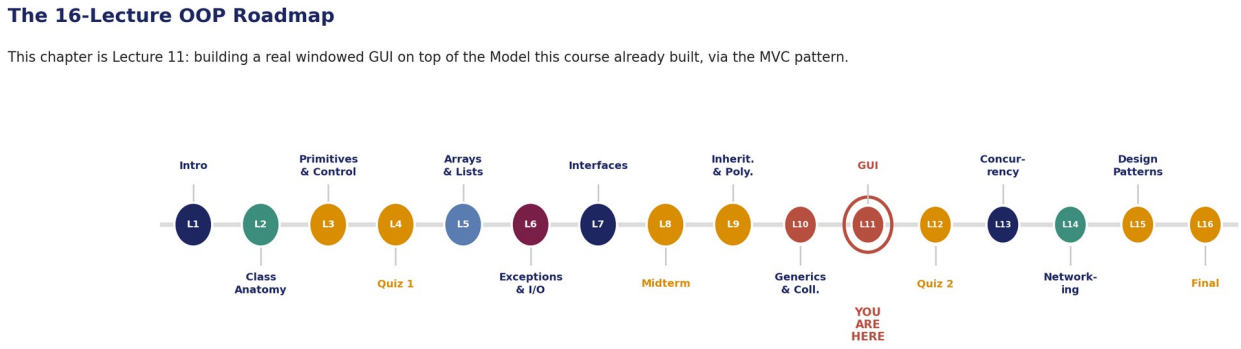


Figure 11.1 — The 16-lecture OOP roadmap. This chapter is Lecture 11: a real windowed GUI built on the same Library/LibraryItem Model this course has built since Chapter 5, via the Model-View-Controller pattern.

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

A console script like library_demo.py is a program IN CONTROL: it decides, line by line, what happens next, and calling 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 main() finishes building the window and calls root.mainloop(), the program's own code stops driving anything. Tkinter's event loop takes over, sits idle until the user does something (clicks a button, types into an entry, 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 function anywhere that describes the whole run from start to finish the way a plain script's top-level code used to.

> **root.mainloop() never returns during an ordinary run**
>
> Tkinter's event loop calls a callback, waits for it to return, then goes back to waiting for the next event -- it does not multitask your callback 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 (get_all_items(), a copy-returning accessor) to make it usable from a View. Every other Model method Chapters 5-10 already wrote -- add_item(), remove_item(), find_by_title(), borrow(), return_item() -- is reused completely unchanged (Figure 11.2).

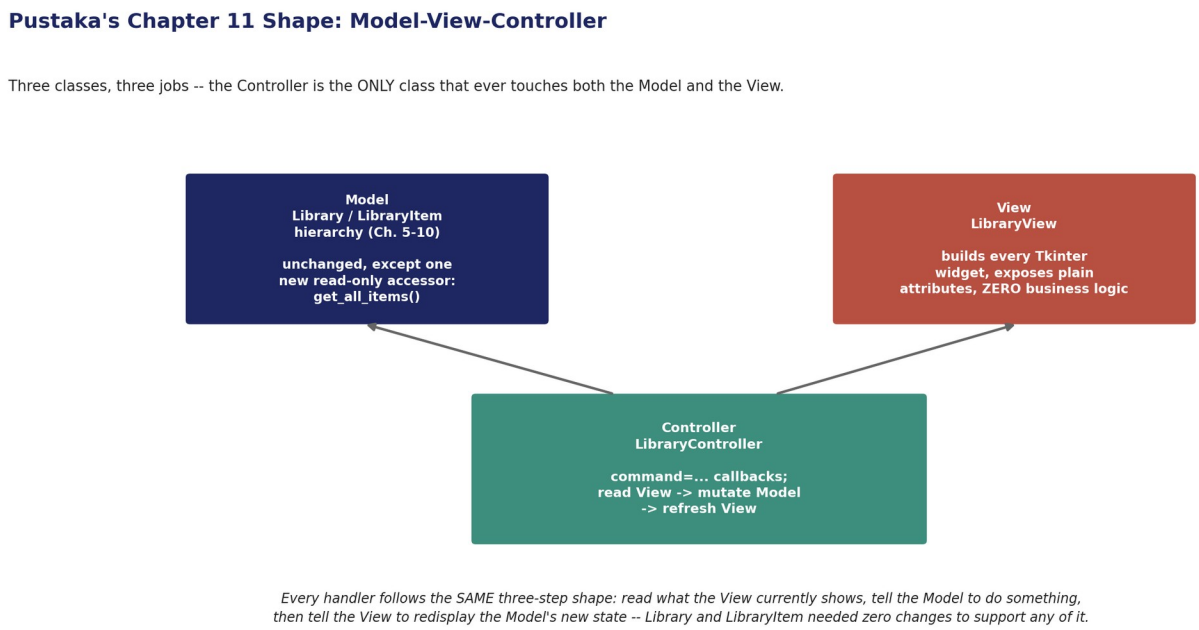


Figure 11.2 — Pustaka's Chapter 11 shape: Model, View, and Controller, with the Controller as the only class touching both of the other two.

> **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 widgets 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 widgets. 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 Tkinter widget in its constructor -- a Listbox for the items, four Entry widgets for adding a book, four ttk.Buttons, and a status Label -- and exposes each one as a plain public attribute (Python has no equivalent of a private field enforced by the language, so the View simply relies on the same naming-convention discipline every prior chapter's classes already use). It calls nothing on Library, and it contains no method that decides what borrowing, adding, or removing actually means; its only substantive method besides construction is refresh(items), which simply redraws whatever list of items it is handed.

library_view.py (excerpt)

```
class LibraryView:
    def __init__(self, root: tk.Tk) -> None:
        self.item_listbox = tk.Listbox(list_frame, ...)
        self.borrow_button = ttk.Button(side, text="Borrow Selected")
        self.return_button = ttk.Button(side, text="Return Selected")
        self.remove_button = ttk.Button(side, text="Remove Selected")
        self.status_label = tk.Label(center, text="Ready.", anchor="w")
        # ... title_entry/author_entry/isbn_entry/year_entry, add_button ...
        self._current_items: list[LibraryItem] = []

    def selected_item(self) -> LibraryItem | None:
        selection = self.item_listbox.curselection()
        if not selection:
            return None
        return self._current_items[selection[0]]

    def refresh(self, items: list[LibraryItem]) -> None:
        previously_selected = self.selected_item()
        self._current_items = items
        self.item_listbox.delete(0, tk.END)
        for item in items:
            self.item_listbox.insert(tk.END, item.describe())
        if previously_selected is not None:
            for index, item in enumerate(items):
                if item is previously_selected:
                    self.item_listbox.selection_set(index)
                    break

    def set_status(self, message: str) -> None:
        self.status_label.config(text=message)
```

A Listbox only holds display strings -- the View keeps its own parallel list of the real objects

Unlike JavaFX's ListView<LibraryItem> (which can hold the real objects directly, via a cell factory that calls describe() for display), Tkinter's Listbox only ever stores plain strings. selected_item() bridges that gap: _current_items is kept in lockstep with whatever refresh() last displayed, so a selected row INDEX from the Listbox can be mapped back to the real LibraryItem object behind it -- the same describe() method every LibraryItem already exposes (Chapters 7-9) supplies the display string, exactly as it does on the Java track.

11.5 The Controller: Wiring Events

LibraryController is the only class in this chapter that imports both Library and LibraryView. Its constructor calls _wire_events() (attaching one command= callback per button) and then _refresh_view() 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 entry fields, tell the Model to do something, then call _refresh_view() 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.

library_controller.py (excerpt)

```
class LibraryController:
    def __init__(self, library: Library, view: LibraryView) -> None:
        self.library = library
        self.view = view
        self._wire_events()
        self._refresh_view()

    def _wire_events(self) -> None:
        self.view.add_button.config(command=self._handle_add_book)
        self.view.borrow_button.config(command=self._handle_borrow_selected)
        self.view.return_button.config(command=self._handle_return_selected)
        self.view.remove_button.config(command=self._handle_remove_selected)

    def _handle_borrow_selected(self) -> None:
        selected = self.view.selected_item()
        if selected is None:
            self.view.set_status("Borrow failed: no item selected.")
            return
        ok = selected.borrow()
        if ok:
            self.view.set_status(f"Borrowed: {selected.title}")
        else:
            self.view.set_status(
                f"Borrow refused: {selected.title} is unavailable "
                "(already on loan, or a reference-only item)."
            )
        self._refresh_view()

    def _refresh_view(self) -> None:
        self.view.refresh(self.library.get_all_items())
```

Tkinter's command= callback takes NO arguments -- unlike JavaFX's setOnAction(...)

A JavaFX setOnAction(e -> handler()) lambda is always handed an ActionEvent object, even when the handler never uses it (Section 11.5 of the Java-track book). Tkinter's command=self._handle_borrow_selected passes no event object at all -- config(command=...) expects a plain zero-argument callable, full stop, and a handler written to expect an argument (def _handle_borrow_selected(self, event):) would raise a TypeError the very first time Tkinter tried to call it with zero arguments. This is a real, easy-to-make mistake for anyone coming from JavaFX (or from any callback API that does pass an event object) to this chapter's Tkinter track.

11.6 The Application Entry Point: library_app.py

library_app.py's main() function is this program's entry point: 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 then calls root.mainloop(). After that call, control passes to Tkinter's own event loop, and nothing else in main() runs during an ordinary interactive session -- every subsequent line of behavior comes from a Controller handler responding to a real click.

library_app.py (excerpt)

```
def main() -> None:
    library = Library()
    library.add_item(Book("Clean Code", "Robert C. Martin", "9780132350884",
2008))
    library.add_item(DVD("The Imitation Game", "9900000000001", 114))
    library.add_item(ReferenceBook("Encyclopedia of Computer Science",
                                   "Ralston & Reilly", "9780123456789", 2003))
    library.add_item(Book("Algorithms", "Robert Sedgewick", "9780321573513",
2011))

    root = tk.Tk()
    view = LibraryView(root)
    LibraryController(library, view)

    root.mainloop()

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

root.mainloop() and Tkinter's own event loop

mainloop() is what actually starts Tkinter's event processing: it begins waiting for X11/window-system events (clicks, key presses, window-close requests) and dispatches each one to whichever widget's command= callback (or other binding) was registered for it. main() calling root.mainloop() and then having nothing execute afterward is normal and expected -- there is no line of code in this program that runs the equivalent of a while(true) loop; Tkinter supplies that loop itself, on the calling thread, and the program only ever reacts.

11.7 Putting It Together: Verifying a GUI Actually Runs

A console chapter's script produces a text transcript that is trivially reproducible: run it again, get the identical print() lines. A GUI has no equivalent text output by default, so this chapter's library_app.py

additionally supports a scripted verification path, gated behind a --verify command-line flag (Python has no built-in system-property mechanism the way the JVM does, so a plain command-line flag read from sys.argv is the natural substitute here -- the difference is only in HOW the flag is passed, not in what the verification actually proves), that a normal interactive run never takes. When that flag is present, root.after(200, ...) schedules run_scripted_verification(), which saves a screenshot of the freshly opened window, selects the first list row PROGRAMMATICALLY (view.item_listbox.selection_set(0)), then calls view.borrow_button.invoke() -- Tkinter's own API for invoking a button exactly as a real mouse click would, running through the identical command= callback a click dispatches to -- and saves a second screenshot afterward.

library_app.py -- run_scripted_verification() (excerpt)

```
def run_scripted_verification(root: tk.Tk, view: LibraryView) -> None:
    save_snapshot(root, "gui_screenshot_1_initial.png")

    view.item_listbox.selection_set(0)
    view.borrow_button.invoke()   # exact same code path as a real mouse click
    print(f'status now reads: "{view.status_label.cget("text")}"')

    save_snapshot(root, "gui_screenshot_2_after_borrow.png")
    root.destroy()

def save_snapshot(root: tk.Tk, file_name: str) -> None:
    root.update_idletasks()
    root.update()
    left = root.winfo_rootx()
    top = root.winfo_rooty()
    right = left + root.winfo_width()
    bottom = top + root.winfo_height()
    screenshot = ImageGrab.grab(bbox=(left, top, right, bottom))
    screenshot.save(file_name)
```

Button.invoke() does not simulate or describe a click; it runs the identical command= callback Tkinter 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

Python has no separate compile step, but this chapter needs a specific interpreter for a different reason than Chapter 10's PEP 695 syntax: Tkinter itself is only usable through the python3-tk system package, and, in this course's environment, that package is compiled against python3.12 specifically -- neither the default python3 (3.11) nor python3.13 has a working tkinter available at all. This is the same python3.12 recommendation Chapter 10 already established for PEP 695 syntax, now for a second, independent reason -- always run this chapter's code with python3.12 explicitly, never with a bare python3.

```
$ python3.12 library_app.py

# scripted verification run (produces the two screenshots in Appendix 11.A):
$ python3.12 library_app.py --verify
```

python3.13 has no installable tkinter package in this environment

Chapter 10 established that python3.12 or python3.13 are both required for PEP 695 syntax; this chapter narrows that further. python3-tk (the apt package providing tkinter) is compiled specifically against python3.12 in this course's environment, and no python3.13-tk package exists to install at all -- import tkinter under python3.13 fails with ModuleNotFoundError, not because tkinter's code is missing, but because no matching compiled package was ever available to install. From this chapter on, the Python track's GUI code requires python3.12 specifically, not merely "3.12 or newer".

Verifying a real GUI without a physical screen

This chapter's code was run in a sandboxed environment with no physical display attached, using Xvfb, a virtual X11 display server: xvfb-run -a python3.12 library_app.py --verify gives Tkinter 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 Tkinter code, the same event dispatch, the same rendering pipeline a desktop run would use -- only the display itself is virtual, not simulated. That requirement is about the sandbox having no monitor at all, not about the screenshot mechanism itself: save_snapshot() (Section 11.7) captures via Pillow's PIL.ImageGrab.grab(), which needs no virtual display of its own on an ordinary desktop -- a --verify run on a real Windows, macOS, or Linux machine screenshots the actual visible window directly.

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 widgets, 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 (get_all_items(), a copy-returning accessor) to become usable from a GUI View -- every borrow()/return_item()/remove_item_or_raise() call is reused completely unchanged.
- Event-driven programming inverts a console script's control flow: Tkinter's own event loop, not the programmer's code, decides when a handler runs, calling back into exactly the callable wired with command= for whichever widget the user just interacted with.
- Tkinter's command= callback takes NO arguments at all, unlike JavaFX's setOnAction(...), which always passes an ActionEvent -- the same three-step handler shape (read View, mutate Model, refresh View) still applies on both tracks, but the callback's own signature differs.
- Button.invoke() 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.

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 library_view.py's Listbox could be swapped for a Treeview, or library_view.py itself replaced by an entirely different toolkit, without changing a single line of library.py.

11.11 Review Questions

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

1. LibraryController._handle_remove_selected() catches ItemNotFoundError even though the ISBN it passes to remove_item_or_raise() came directly from a LibraryItem already sitting in the View's own list. Explain why that except block still belongs in the code, and why the branch inside it should never actually run in practice.
2. Suppose LibraryView exposed library directly (a public attribute) instead of only exposing its own widgets. 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.borrow_button.invoke() is used in this chapter's verification run instead of directly calling library.find_by_title(...).borrow() from the verification code. Explain what a direct Model call like that would fail to prove that invoke() does prove.
4. Tkinter's command= callback receives no arguments, while JavaFX's setOnAction(...) always receives an ActionEvent. Rewrite _handle_borrow_selected as though it needed to accept an event argument anyway (def _handle_borrow_selected(self, event=None):), and explain why giving the parameter a default value is what keeps command=self._handle_borrow_selected working unchanged.
5. Rewrite _handle_borrow_selected() so that, instead of calling selected.borrow() directly, it first checks selected.is_on_loan 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 library_item.py, book.py, dvd.py, reference_book.py, item_not_found_error.py, pair.py, library_utils.py, library.py, library_view.py, library_controller.py, and library_app.py files (Code/Python/Chapter11/, provided alongside this book) were run under Xvfb on python3.12 (with tkinter 8.6) before this chapter was written, using the --verify 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).

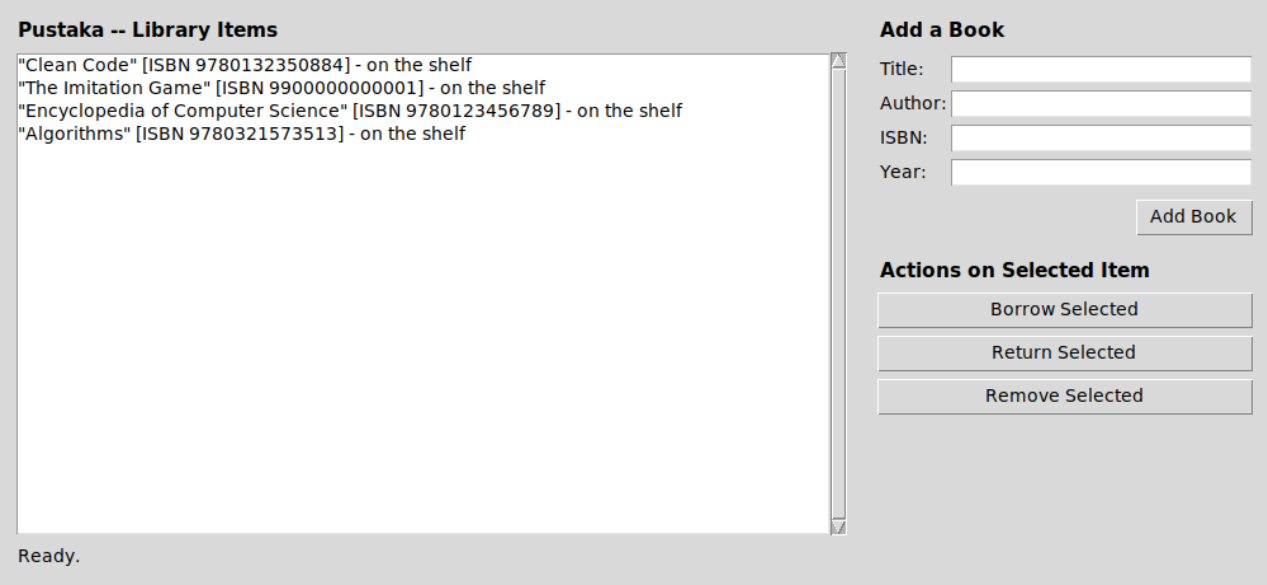


Figure 11.4 — Screenshot 1: the window immediately after opening. All four items show "on the shelf"; nothing is selected yet.

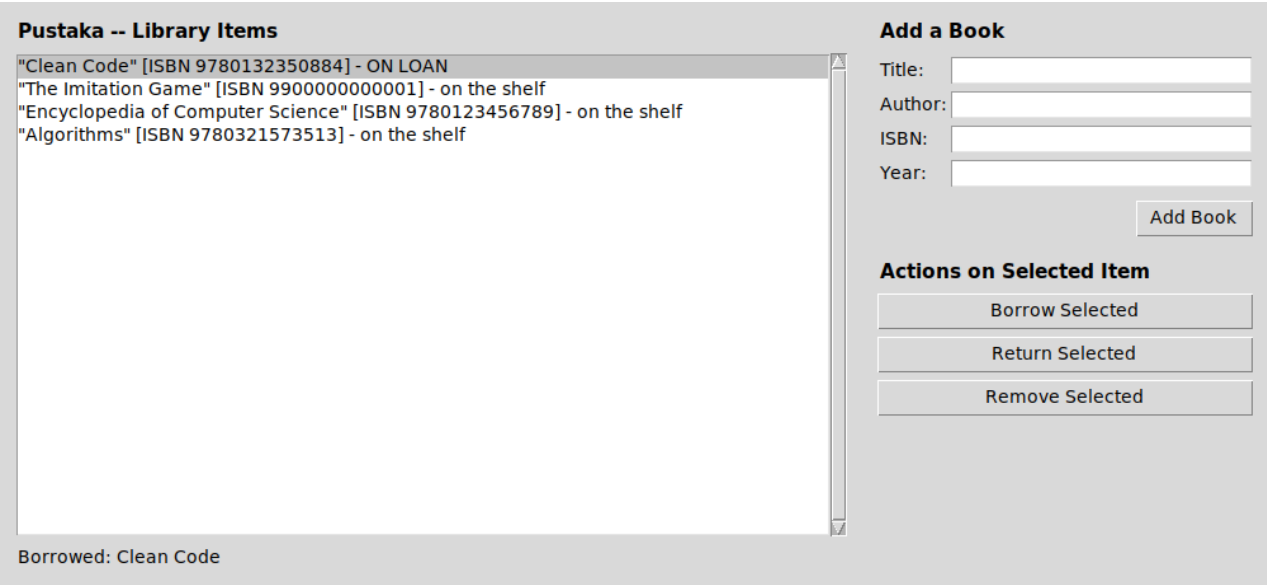


Figure 11.5 — Screenshot 2: after the scripted run selected row 0 and invoked Borrow Selected. "Clean Code" is highlighted and now reads "ON LOAN"; the status label reads "Borrowed: Clean Code".

```
$ xvfb-run -a python3.12 library_app.py --verify
--- 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] Python Software Foundation. "tkinter — Python interface to Tcl/Tk." Python 3 documentation. https://docs.python.org/3/library/tkinter.html (accessed August 2026).

[2] Python Software Foundation. "An Introduction to Tkinter — Events and Bindings." Python 3 documentation. https://docs.python.org/3/library/tkinter.html#bindings-and-events (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] X.Org Foundation. "Xvfb — virtual framebuffer X server." X.Org manual pages. https://www.x.org/releases/X11R7.7/doc/man/man1/Xvfb.1.xhtml (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 attributes 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 an idiomatic join 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 ABC 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 function bounded to an orderable type, build an index using a dict, 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), numeric types and control flow (Lecture 3), Quiz 1 (Lecture 4), lists and string building (Lecture 5), exceptions and file I/O (Lecture 6), abc.ABC and typing.Protocol (Lecture 7), and the Midterm itself (Lecture 8). Lecture 9 built directly on Lecture 7's ABC and Protocol with true inheritance — subclassing, super(), method overriding, and dynamic dispatch that Python resolves at runtime through its own duck-typing rules. Lecture 10 added Python's generics (TypeVar, Generic) and its built-in collections, letting Pustaka's LibraryItem hierarchy be stored, sorted, and looked up in a type-annotated list, set, or dict instead of a plain list. Lecture 11 took Library out of the console entirely, building LibraryView and LibraryController on top of Tkinter'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).

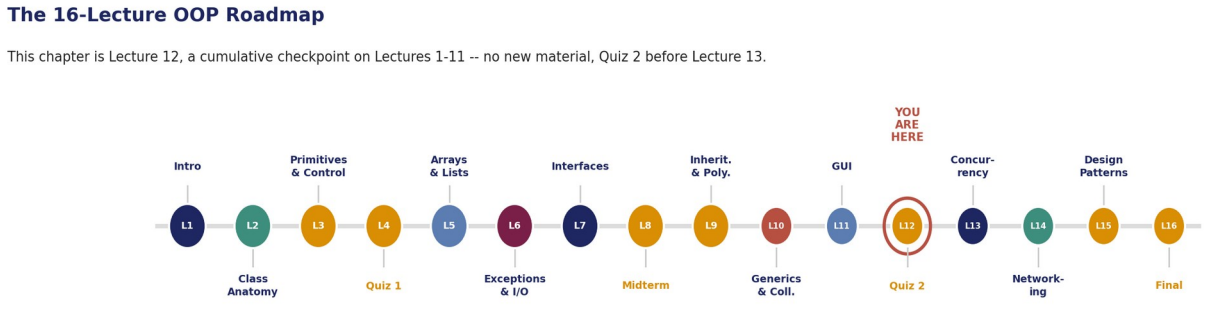


Figure 12.1 — This chapter sits at Lecture 12 in the sixteen-lecture OOP roadmap: a checkpoint, not a new topic, Quiz 2 before Lecture 13.

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 -- attributes, constructor, encapsulation
2	BMI Category	L3 -- control flow, boundary conditions
3	CSV Row Formatter	L5 -- string building, lists
4	Negative Balance Guard	L6 -- a custom exception class, a transaction log file
5	Shape Hierarchy	L7 -- ABC, a concrete method built on an abstract one
6	Vehicle & Car	L9 -- inheritance, polymorphism, super().describe()
7	Generic max_of	L10 -- a bounded TypeVar, a dict-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 .py file 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 Attributes, Constructors & Encapsulation Review

Problem 1 *[Easy]*. Define a class Rectangle(width, height), storing only the two dimensions, with get_width(), get_height(), get_area() (width * height), and get_perimeter() (2 * (width + height)).

Hint

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

12.3.2 Numeric Types & Control Flow Review

Problem 2 *[Easy]*. Write a function bmi_category(weight_kg, height_m) that computes bmi = weight_kg / (height_m ** 2) 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, using elif, and return as soon as one 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 Lists & String Building

Problem 3 *[Medium]*. Write a function format_csv_row(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 idiomatically rather than with repeated string concatenation in a loop.

> **Hint**
>
> Build a new list where each field is either left unchanged or wrapped in escaped double quotes (based on whether "," appears in the field), then call ",".join(...) on that list once — exactly the join-at-the-end idiom Chapter 8's format_roster() already used.

12.3.4 Exception Handling & File I/O

Problem 4 *[Medium]*. Define a custom exception class NegativeBalanceError(Exception), and a class Account(opening_balance) with a method withdraw(self, amount) that raises a NegativeBalanceError (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,balance_after" to a transaction log file.

> **Hint**
>
> Check the insufficient-funds condition and raise BEFORE touching the balance attribute or opening the file — a failed withdrawal must leave both completely untouched. Once the check passes, update the balance attribute first, then use with open(path, "a") as f: to append the log line, exactly the pattern Chapter 6's save_to_file used for Pustaka and Chapter 8's append_score_record used for scores.

12.3.5 ABC & Protocol

Problem 5 *[Medium]*. Define an abstract class Shape(ABC) with an abstract method area(self), and a CONCRETE method describe(self) built entirely from area() (e.g. returning something like "This shape covers 28.27 square units."). Then define two concrete subclasses: Circle(radius) and Square(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 Employee.annual_pay() being a concrete method built on top of the abstract monthly_pay().

12.3.6 Inheritance & Polymorphism

Problem 6 *[Medium]*. Define a class Vehicle(make, model) with a method describe(self) returning "make model". Then define a subclass Car(make, model, door_count) that OVERRIDES describe() to call super().describe() and append the door count, e.g. "Toyota Corolla (4 doors)". Finally, build a list containing a mix of plain Vehicle and Car objects, and call describe() on each in a loop.

> **Hint**
>
> Car.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 the mixed list is polymorphic dispatch: even though every element is conceptually a Vehicle, each call to describe() resolves at runtime to whichever class the actual object belongs to — Python's duck typing makes this automatic, but calling super() is still a discipline you must apply yourself.

12.3.7 Generics & Collections

Problem 7 *[Hard]*. Write a generic function max_of(items: list[T]) -> T (with T a TypeVar bound to an orderable type) that returns the largest element of items, raising ValueError for an empty list. Then write a function group_by_first_letter(words: list[str]) -> dict[str, list[str]] that groups words into a dict keyed by their (uppercased) first letter.

> **Hint**
>
> Python has no compile-time bound the way Java's <T extends Comparable<T>> does, but declaring T = TypeVar("T", bound=...) documents the same ordering requirement for anyone reading the signature, even though Python only checks it at runtime rather than compile time. For group_by_first_letter, use dict.setdefault(key, []).append(word), or a collections.defaultdict(list), so you never need a separate 'is this key already in the dict' check before appending to a group's list.

12.3.8 GUI & MVC

Problem 8 *[Hard]*. Design a tiny MVC counter application: a Model class CounterModel with increment(), decrement(), and a value property; 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's command to the matching Model method and then refreshing the View's label. Sketch the three classes' attributes 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 tkinter, 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 command callback modifies CounterModel'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 code that runs 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 or function produce the exact expected result for every given test case, including boundary values?	45%
Class, ABC & generic design	Are attributes accessed only through methods, and are the ABC split (Problem 5) and the TypeVar 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 the with statement correctly, and does Problem 8's sketch keep Model, View, and Controller cleanly separated?	15%
Code clarity & runs cleanly	Is the code easy to read, sensibly named, and free of dead code — and does it run 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's Python/FastAPI backend 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 dict-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), string building and lists (L5), custom exceptions and file I/O (L6), ABC (L7), inheritance and polymorphism (L9), generics and collections (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.

Code that runs is not the same as code 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 function and class run with zero errors on every given test case?
- Are attributes accessed only through methods, with no code outside the class reaching in directly?
- 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 attribute and the log file completely untouched when it raises NegativeBalanceError?
- 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 Car.describe() call super().describe() rather than reformatting make and model itself?
- Does Problem 7's max_of() document its ordering requirement with a bound TypeVar, and does group_by_first_letter() avoid a separate membership check before appending 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] Python Software Foundation. "typing — TypeVar and Generic." Python 3 documentation. https://docs.python.org/3/library/typing.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).

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 -- including one honest surprise about how hard this particular bug turned out to be to observe naturally in Python (Figure 13.1).

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

(1) Explain why an operation as simple as count += 1 is not atomic, and describe concretely how two threads can interleave to lose an update. (2) Create and start multiple threading.Threads to run work concurrently, and join() them to wait for every one to finish. (3) Use a threading.Lock to enforce mutual exclusion on shared state, and explain exactly what acquiring a lock blocks other threads from doing. (4) Use a concurrent.futures.ThreadPoolExecutor 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 Python's built-in collections. 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.

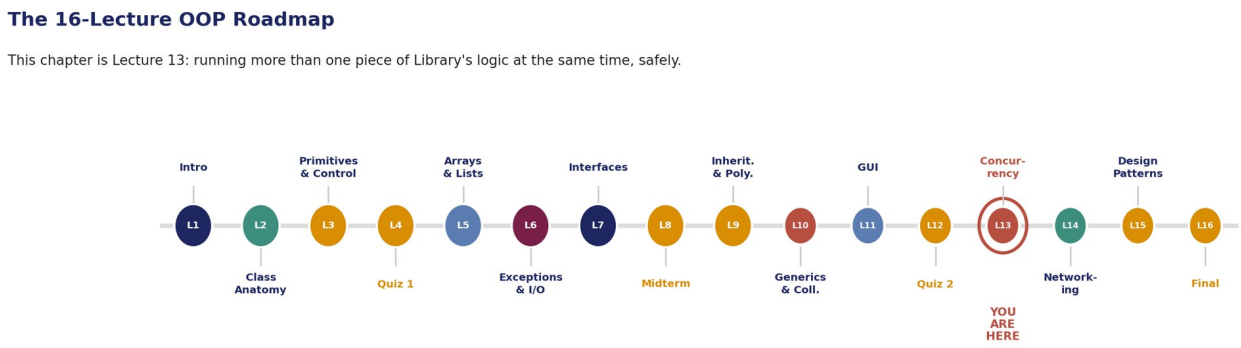


Figure 13.1 — The 16-lecture OOP roadmap. This chapter is Lecture 13: running more than one piece of Library's logic at the same time, safely.

13.2 Why Concurrency: Threads, Shared State, and the GIL

A threading.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. CPython (the reference Python implementation this course uses) additionally has a Global Interpreter Lock (GIL), which allows only

one thread to execute Python bytecode at any single instant -- but the GIL does NOT make a multi-step Python statement atomic, and it does not remove the need for explicit synchronization, as Section 13.3 demonstrates concretely. 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.

borrow_counter.py -- a minimal shared-state contract

```
@runtime_checkable
class BorrowCounter(Protocol):
    def increment(self) -> None: ...
    def get_count(self) -> int: ...
```

Two classes satisfy this Protocol structurally, exactly the way Chapter 7's Book and DVD satisfied Renewable. 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

self._count += 1 looks like a single, indivisible operation in Python source code. It is not. CPython performs it as three separate bytecode-level steps: load the current value of _count, add one to it, and store the new value back. The GIL protects each individual bytecode instruction, not a whole Python statement -- so nothing prevents the interpreter from switching to another thread between those three steps, and when it does, an increment is silently lost (Figure 13.2).

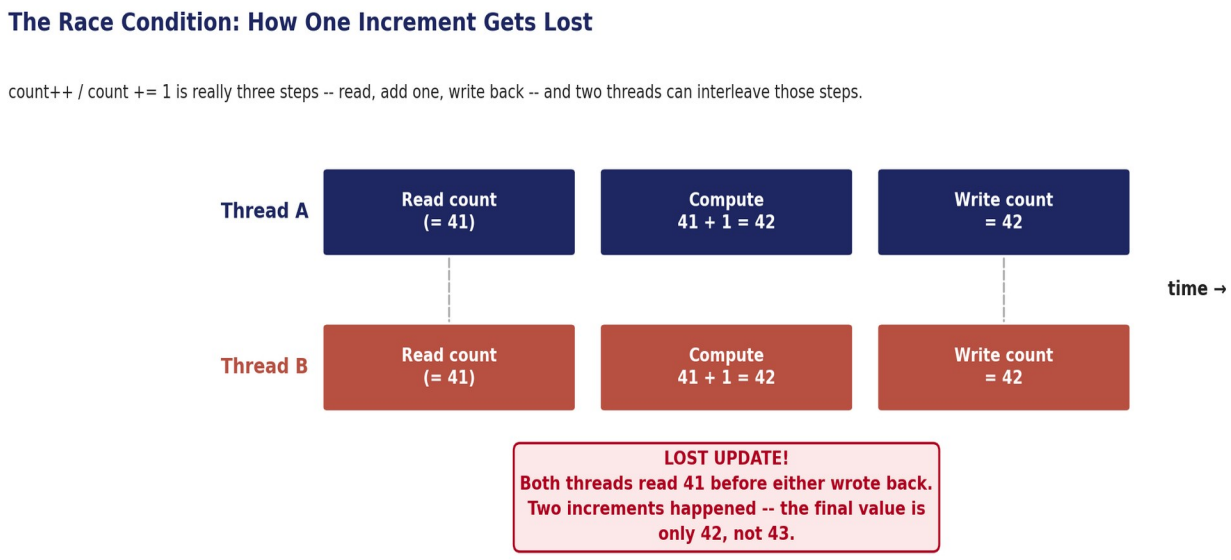


Figure 13.2 — Two threads both read count as 41 before either writes back; both compute 42 and write 42. One of the two increments never happened, and nothing in the program raised an error to say so.

unsafe_borrow_counter.py -- the naive, buggy implementation

```
class UnsafeBorrowCounter:
    def __init__(self) -> None:
        self._count = 0

    def increment(self) -> None:
        current = self._count
        time.sleep(0)  # widens the race window -- see the note below
        self._count = current + 1

    def get_count(self) -> int:
        return self._count
```

Honesty note: this book's own testing needed to widen the window to see the race

A plain self._count += 1, run across 20 threads and 20 million total increments during this book's own testing, produced ZERO lost updates on the sandboxed machine this book was written on -- CPython 3.12's eval-loop check interval simply did not happen to land inside the load/add/store sequence often enough in that environment. The race condition is still completely real (it is documented CPython behavior, confirmed by the Python track's own Reference [2] below, not a claim invented for this book) -- it was just too rare to show up reliably inside a short demo on that one machine. The time.sleep(0) above is inserted deliberately, between the read and the write, to widen that interleaving window so the SAME real hazard becomes visible on demand, exactly the way a slow-motion camera reveals a real collision that happens too fast to see live. It does not fabricate a hazard that was not already there, and Section 13.9's own Profitina discussion explains why the underlying hazard still matters in production code that has no such artificial delay.

13.4 Synchronization: Mutual Exclusion with threading.Lock

A threading.Lock object can be acquired by only one thread at a time; a with self._lock: block acquires the lock on entry and releases it automatically on exit, even if an exception is raised inside. While one thread holds the lock, every other thread that tries to acquire the SAME Lock object blocks -- it simply waits -- until the lock is released. That turns "load, add one, store back" into an operation that behaves atomically from every other thread's point of view, even though it is still three bytecode-level steps underneath -- the same fix as the Java track's synchronized keyword, just spelled as an explicit object rather than a keyword on the method itself.

safe_borrow_counter.py -- the fixed implementation

```
class SafeBorrowCounter:
    def __init__(self) -> None:
        self._count = 0
        self._lock = threading.Lock()

    def increment(self) -> None:
        with self._lock:
            self._count += 1

    def get_count(self) -> int:
        with self._lock:
            return self._count
```

get_count() needs the lock too, not just increment()

Locking only increment() would stop two increments from interleaving with EACH OTHER, but a thread calling get_count() 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. Acquiring the same Lock in both methods closes both gaps at once.

13.5 A Thread Pool: ThreadPoolExecutor

Creating and starting one threading.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. concurrent.futures.ThreadPoolExecutor 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

```
with ThreadPoolExecutor(max_workers=4) as pool:
    futures = [
        pool.submit(borrow_worker, counter, ITERATIONS)
        for _ in range(KIOSK_COUNT)
    ]
    for future in futures:
        future.result()
```

> **The with block does not return until every submitted task finishes**
>
> Exiting the with ThreadPoolExecutor(...) as pool: block calls pool.shutdown(wait=True) implicitly, which blocks until every already-submitted task completes -- and calling future.result() on each Future additionally re-raises any exception that task raised, rather than letting it vanish silently in a background thread.

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 get_all_items() 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 attributes 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.

borrow_worker.py -- simulates one patron kiosk

```
def borrow_worker(counter: BorrowCounter, iterations: int) -> None:
    for _ in range(iterations):
        counter.increment()
```

13.7 The Driver Program: Three Demos, Back to Back

concurrency_demo.py simulates 8 patron kiosks. Demo 1 runs UnsafeBorrowCounter with a SMALLER iteration count (2,000 per kiosk) than Demos 2-3, because each unsafe increment now sleeps briefly to make the race reliably visible (see Section 13.3's honesty note) -- so an unsafe run at the full 200,000-per-kiosk workload would take far too long. Demo 2 runs the full 200,000-per-kiosk workload against SafeBorrowCounter with 8 manually created Threads; Demo 3 runs it a third time against SafeBorrowCounter, but through a 4-worker ThreadPoolExecutor instead of 8 manually managed Threads.

concurrency_demo.py -- running a demo with manual threads (abridged)

```
def run_with_manual_threads(counter, iterations: int) -> int:
    kiosks = [
        threading.Thread(target=borrow_worker, args=(counter, iterations))
        for _ in range(KIOSK_COUNT)
    ]
    for kiosk in kiosks:
        kiosk.start()
    for kiosk in kiosks:
        kiosk.join()
    return counter.get_count()
```

start() begins a thread; calling the target function directly does not

kiosk.start() asks the operating system to begin a genuinely new thread of execution, which then calls the target function on its own. Calling borrow_worker(counter, iterations) directly, without going through Thread.start(), would just execute the loop on the CURRENT thread, sequentially, with no concurrency at all -- a common, easy mistake that produces code that runs and even produces the right number, for the wrong reason.

13.8 Compiling and Running Your Program

Python has no compile step, but tkinter's Chapter 11 lesson about needing python3.12 explicitly does not apply here -- this chapter's threading and concurrent.futures modules are part of the standard library on any supported Python 3 version. python3.12 is still used below purely for consistency with the rest of this book's Python track.

```
$ python3.12 concurrency_demo.py
```

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 Python/FastAPI 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 can silently under-count under real production load, the same underlying hazard UnsafeBorrowCounter demonstrates here (with an artificial delay only because this book's own testing happened not to trigger it naturally in a short demo -- production code under real, sustained, unpredictable load has no such guarantee of safety). The fix in production code is the same idea this chapter teaches at a much smaller scale: either explicit locking around the shared state, or (more often, at Profitina's scale) a dedicated concurrency-safe primitive from the platform's standard library or its async framework 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

- self._count += 1 is really three steps -- load, add one, store back -- and is NOT atomic, even under CPython's GIL, which protects individual bytecode instructions rather than whole Python statements.
- This chapter's own testing found the natural race surprisingly rare on one sandboxed machine, and disclosed exactly why and exactly what artificial change (a brief sleep) was used to make it reliably observable -- see Section 13.3.

- A threading.Lock, acquired with a with block, enforces mutual exclusion: every other thread trying to acquire the SAME Lock blocks until it is released.
- A ThreadPoolExecutor runs many short tasks over a small, fixed, reusable set of worker threads, instead of one Thread object per task -- and calling .result() on each Future re-raises any exception a task encountered.
- 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 attributes included.

A bug that never shows up under a single thread, and that no linter has anything 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 locks both increment() and get_count(). Explain, in your own words, what could go wrong if only increment() acquired the lock and get_count() were left as a plain, unlocked method.
2. Rewrite borrow_worker() so that, instead of calling counter.increment() in a loop, it builds its own local total variable 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. concurrency_demo.py's Demo 3 uses ThreadPoolExecutor(max_workers=4) for 8 kiosks. Explain what happens to the 5th, 6th, 7th, and 8th borrow_worker 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 a Lock needed here? Explain why or why not, referring back to what a Lock actually restricts.
5. Section 13.3's honesty note explains that the race condition did not appear reliably without an artificial change on this book's testing machine. Explain, in your own words, why a race condition can still be considered a real production hazard even if it does not appear on every single run, or even on every single machine.

Appendix 13.A — Execution Verification Log

The complete borrow_counter.py, unsafe_borrow_counter.py, safe_borrow_counter.py, borrow_worker.py, and concurrency_demo.py files (Code/Python/Chapter13/, provided alongside this book) were run on python3.12 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 concurrency_demo.py will very likely show a DIFFERENT number of lost updates than the run captured below. What is NOT expected to vary is the qualitative result: Demo 1 (unsafe) reliably fell short of its expected count across repeated runs during this book's own testing, while Demo 2 and Demo 3 (both safe) matched their expected count exactly, every single time.

```
$ python3.12 concurrency_demo.py
Simulating 8 patron kiosks.

--- Demo 1: UnsafeBorrowCounter (no synchronization) ---
(2000 borrow operations per kiosk -- fewer than Demos 2-3, since each unsafe
increment sleeps briefly to make the race visible; see
unsafe_borrow_counter.py.)
Mathematically correct final count: 16000
Actual final count:                 2012
Lost updates:                       13988  <-- a real race condition, not a typo

--- Demo 2: SafeBorrowCounter (threading.Lock) ---
(200000 borrow operations per kiosk.) Mathematically correct final count:
1600000
Actual final count:                 1600000
Matches expected:                   True

--- Demo 3: SafeBorrowCounter via a thread pool (ThreadPoolExecutor) ---
Actual final count:                 1600000
Matches expected:                   True
```

References

[1] Python Software Foundation. "threading — Thread-based parallelism." Python 3 documentation. https://docs.python.org/3/library/threading.html (accessed August 2026).

[2] Python Software Foundation. "Thread State and the Global Interpreter Lock." Python/C API Reference Manual. https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock (accessed August 2026).

[3] Python Software Foundation. "concurrent.futures — Launching parallel tasks." Python 3 documentation. https://docs.python.org/3/library/concurrent.futures.html (accessed August 2026).

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 threading.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 socket, accept() incoming connections, and read from and write to a socket's data using makefile(). (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 threading.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 library_server.py process and a library_client.py process, running as two separate programs (potentially on two separate machines), that only ever talk to each other over a network connection.

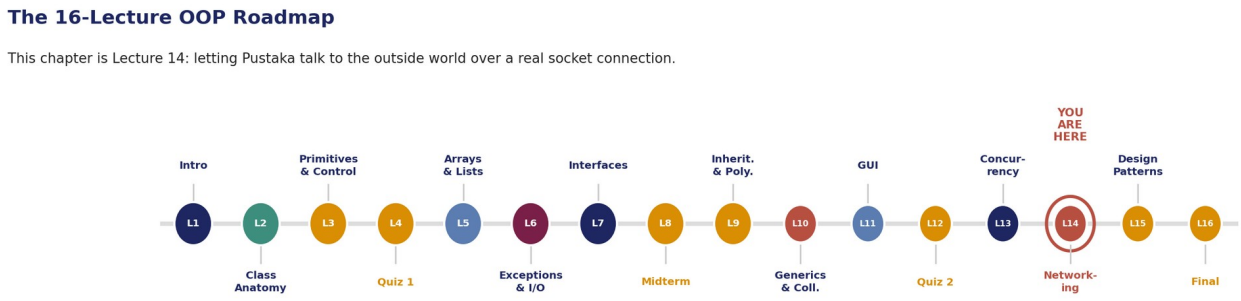


Figure 14.1 — The 16-lecture OOP roadmap. This chapter is Lecture 14: letting Pustaka talk to the outside world over a real socket connection.

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

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

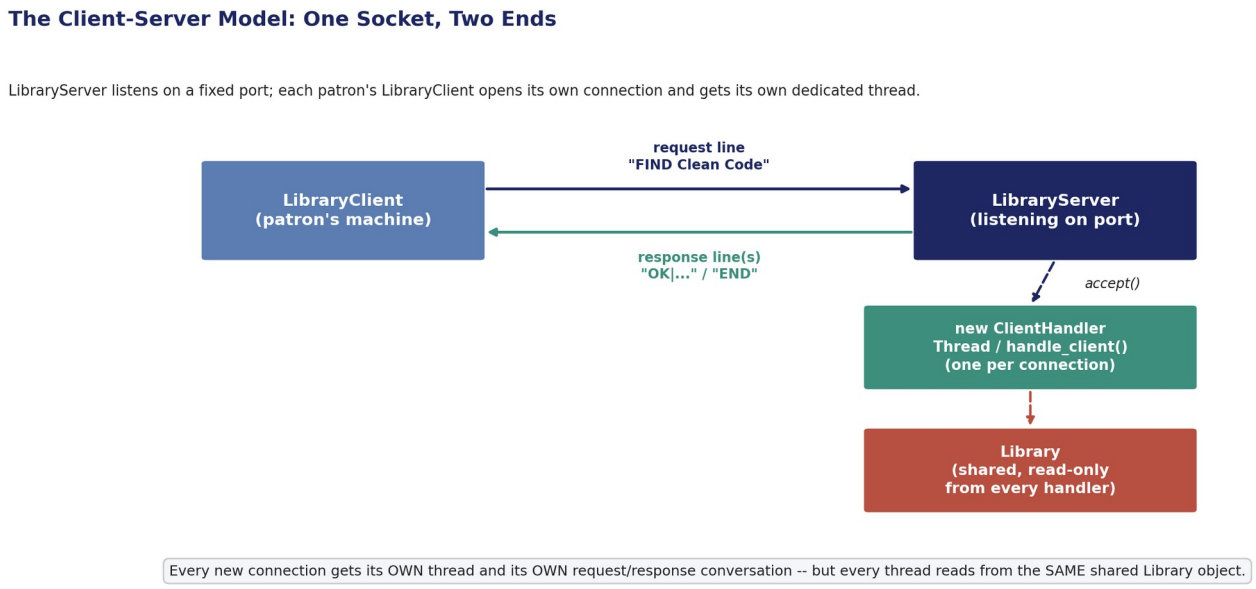


Figure 14.2 — LibraryServer listens on a fixed port; each patron's LibraryClient opens its own connection and gets its own dedicated thread, but every thread reads from the same shared Library object.

A socket is a two-way street, not a one-way pipe

The same connected socket a client uses to SEND a request line is also the object it reads the server's response FROM -- calling makefile("w") and makefile("r") on it gives two views onto the same underlying connection, not two separate connections.

14.3 Opening a Listening Socket

library_server.py -- opening the socket and accepting connections (excerpt)

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_socket.bind((HOST, PORT))
    server_socket.listen()
    print(f"LibraryServer listening on port {PORT}...")
    while True:
        conn, addr = server_socket.accept()   # blocks until a patron connects
        client_thread = threading.Thread(
            target=handle_client, args=(conn, addr, library))
        client_thread.start()
```

accept() blocks -- and this loop never ends on its own

server_socket.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.

library_server.py -- one dedicated thread per connection (excerpt)

```
def handle_client(conn: socket.socket, addr, library: Library) -> None:
    with conn:
        reader = conn.makefile("r", encoding="utf-8", newline="\n")
        writer = conn.makefile("w", encoding="utf-8", newline="\n")
        for line in reader:
            request = line.rstrip("\n")
            if request.upper() == "LIST":
                for item in library.get_all_items():
                    writer.write(f"ITEM|{item.describe()}\n")
                writer.write("END\n")
                writer.flush()
            # FIND and QUIT branches omitted -- see Code/Python/Chapter14/
```

conn.makefile(...) wraps the raw socket in a file-like object

makefile("r", ...) and makefile("w", ...) give text-mode, line-oriented reading and writing over a socket -- Python's equivalent of wrapping Java's Socket streams in a BufferedReader/PrintWriter -- and the with conn: block closes the underlying connection cleanly however this function exits, the same discipline Chapter 6 taught for files, applied here to a network connection instead.

14.5 One Thread Per Client: Networking Meets Chapter 13

library_server.py hands every accepted connection to a brand-new threading.Thread running handle_client(), 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 threading.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 handler at once. Section 14.9'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 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 handler here only calls get_all_items() and find_by_title() -- both of which only READ Library's fields. If a future version of this protocol ever added a BORROW command that called add_item() or remove_item() from inside handle_client(), Chapter 13's whole lesson would apply again immediately: multiple threads mutating the same list and dict without synchronization would risk the exact same kind of lost update BorrowCounter demonstrated.

14.6 The Client Side: library_client.py

library_client.py -- sending one command, reading one response block (excerpt)

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((host, port))
    writer = sock.makefile("w", encoding="utf-8", newline="\n")
    reader = sock.makefile("r", encoding="utf-8", newline="\n")
    writer.write(command + "\n")
    writer.flush()   # unlike Java's PrintWriter(..., autoFlush=True), Python's
                     # makefile() writer needs an explicit flush()
    for line in reader:
        text = line.rstrip("\n")
        print(text)
        if text == "END" or text.startswith("OK|") \
           or text.startswith("ERROR|") or text == "BYE":
            break   # this simple protocol's responses are always exactly one
block
```

sock.connect((host, port)) is the CLIENT-side call

Unlike a listening socket, which waits passively for connections, calling connect() actively INITIATES a connection attempt to that address -- it either succeeds (the call returns) or raises an OSError (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: library_server.py must already be running before library_client.py can connect to it. Nothing here needs compiling -- but both programs, like every other Python file in this book's track, are run with python3.12 purely for consistency with the rest of the Python track.

```
$ python3.12 library_server.py &
LibraryServer listening on port 5052...
$ python3.12 library_client.py localhost 5052 LIST
$ python3.12 library_client.py localhost 5052 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 Python/FastAPI backend plays exactly library_server.py'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 library_client.py'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 listening socket bound to a fixed port and calls accept() in a loop, which blocks until a client connects; a client calls connect((host, port)) to actively initiate a connection.
- Both ends of an established connection read from and write to the SAME socket's data -- makefile("r") and makefile("w") 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.
- library_server.py hands each accepted connection to its own threading.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 handler here only READS the shared Library; that is exactly why no 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 server_socket.accept() must be called again immediately inside the while True: loop, rather than just once before the loop begins.
2. Suppose handle_client() did NOT wrap the connection in a with conn: block. 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 handler 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. library_client.py exits after receiving exactly one response block. Would it be a client-side change or a server-side change to let a single 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.py, dvd.py, item_not_found_error.py, library.py, library_demo.py, library_item.py, library_utils.py, pair.py, reference_book.py, renewable.py, library_server.py, and library_client.py files (Code/Python/Chapter14/, provided alongside this book) were run on python3.12 before this chapter was written -- with library_server.py running as a genuinely separate process, and multiple library_client.py 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.

```
$ python3.12 library_server.py &
LibraryServer listening on port 5052...

$ python3.12 library_client.py localhost 5052 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

$ python3.12 library_client.py localhost 5052 FIND Clean Code
OK|"Clean Code" [ISBN 9780132350884] - on the shelf

$ python3.12 library_client.py localhost 5052 FIND Nonexistent Book
ERROR|Not found: Nonexistent Book

$ python3.12 library_client.py localhost 5052 QUIT
BYE
```

Two clients connecting at the same instant were both served correctly

This chapter's own testing also launched two library_client.py processes at essentially the same moment (one requesting FIND Clean Code, the other FIND The Imitation Game) against one running library_server.py; 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] Python Software Foundation. "socket — Low-level networking interface." Python 3 documentation. https://docs.python.org/3/library/socket.html (accessed August 2026).

[2] Python Software Foundation. "Socket Programming HOWTO." Python 3 documentation. https://docs.python.org/3/howto/sockets.html (accessed August 2026).

[3] Python Software Foundation. "threading — Thread-based parallelism." Python 3 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 handle_client() dispatch is a Command-like shape, 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.

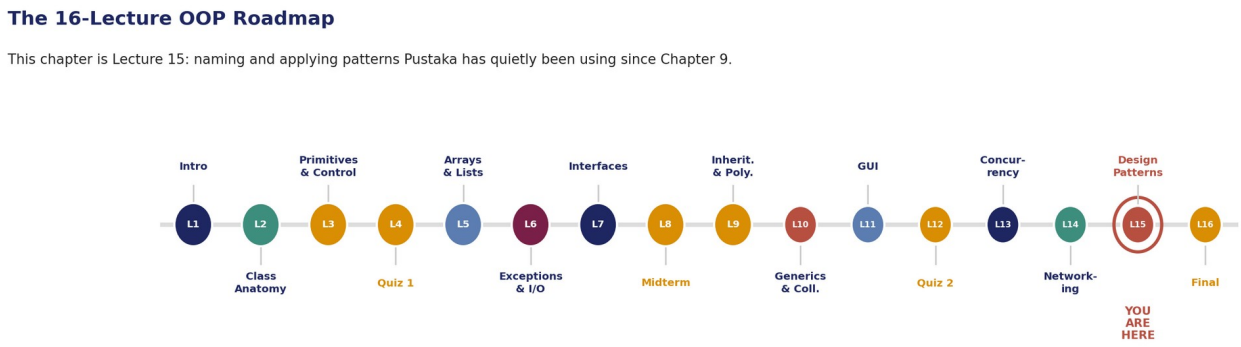


Figure 15.1 — The 16-lecture OOP roadmap. This chapter is Lecture 15: naming and applying patterns Pustaka has quietly been using since Chapter 9.

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 9/10, Library.load_from_file() has read a type-tagged line (BOOK|..., REFBOOK|..., DVD|...) and reconstructed the matching concrete LibraryItem subtype, inline, inside a private _from_line() method. That decision -- "given this tag, which class do I build?" -- is exactly what the Factory Method pattern names: a dedicated function (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.

library_item_factory.py -- the reconstruction logic, extracted (complete)

```
from book import Book
from dvd import DVD
from library_item import LibraryItem
from reference_book import ReferenceBook

def create_from_line(line: str) -> LibraryItem:
    parts = line.rstrip("\n").split("|")
    tag = parts[0]
    if tag == "BOOK":
        return Book(parts[1], parts[2], parts[3], int(parts[4]))
    elif tag == "REFBOOK":
        return ReferenceBook(parts[1], parts[2], parts[3], int(parts[4]))
    elif tag == "DVD":
        return DVD(parts[1], parts[2], int(parts[3]))
    raise ValueError(f"Unknown item type tag: {tag}")
```

Unlike the Java track, which mirrors LibraryUtils's stateless-utility-class shape (a private constructor, only static methods), the Python track expresses Factory Method as a plain module-level function -- Python does not need a class at all just to hold a function with no state, and forcing one here would only be imitating Java's syntax, not its intent. 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. _to_line() -- 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 module, one clear responsibility) argues against merging the two just because they are each other's inverse.

A genuine bug surfaced by this refactor: find_by_isbn() after reload

Extracting create_from_line() required rewriting load_from_file() to call it, and rewriting load_from_file() surfaced a real, previously undetected defect present unchanged since Chapter 9/10: the OLD load_from_file() called self._items.append(self._from_line(line)) directly -- populating the items list, but never the _by_isbn dict. Every item ever reloaded from a saved file was therefore invisible to find_by_isbn(), even though find_by_title() worked correctly (masking the bug in every prior chapter's demo, none of which happened to call find_by_isbn on a reloaded item). The NEW load_from_file() routes every reconstructed item through add_item() instead -- and add_item() 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 tkinter, or any specific presentation technology, exists. The Observer pattern is exactly this: Library holds a list of listeners that satisfy a narrow structural contract, 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.

library_observer.py -- the notification contract (complete)

```python
from typing import Protocol, runtime_checkable

from library_item import LibraryItem

@runtime_checkable
class LibraryObserver(Protocol):
    def on_item_added(self, item: LibraryItem) -> None: ...
    def on_item_removed(self, item: LibraryItem) -> None: ...
```

library.py -- registering observers and notifying them (excerpt)

```python
def __init__(self) -> None:
    self._items: list[LibraryItem] = []
    self._by_isbn: dict[str, LibraryItem] = {}
    self._observers: list[LibraryObserver] = []

def add_observer(self, observer: LibraryObserver) -> None:
    self._observers.append(observer)

def _notify_item_added(self, item: LibraryItem) -> None:
    for observer in self._observers:
        observer.on_item_added(item)

def add_item(self, item: LibraryItem) -> None:
    self._items.append(item)
    self._by_isbn[item.isbn] = item
    self._notify_item_added(item)   # NEW this chapter
```

Because Python's typing.Protocol gives LibraryObserver a STRUCTURAL contract -- the same style Chapter 7's Renewable and Chapter 13's BorrowCounter already used -- ConsoleLogObserver satisfies it simply by defining matching methods, with no explicit inheritance or `implements` keyword required, unlike the Java track's `class ConsoleLogObserver implements LibraryObserver`. Because load_from_file() now routes every reconstructed item through add_item() (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 find_by_isbn 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 add_item(), remove_item(), or load_from_file() -- only a new class satisfying the LibraryObserver Protocol and one add_observer() 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, and CPython's Global Interpreter Lock does NOT protect this gap: a thread can be swapped out at exactly the instant between the None-check and the attribute assignment (Figures 15.2, 15.3 and 15.4).

naive_singleton.py -- the classic, unsynchronized version (complete)

```
class NaiveSingleton:
    _instance: "NaiveSingleton | None" = None

    def __init__(self) -> None:
        pass

    @classmethod
    def get_instance(cls) -> "NaiveSingleton":
        if cls._instance is None:              # (1) check
            cls._instance = NaiveSingleton()    # (2) create and assign
        return cls._instance
```

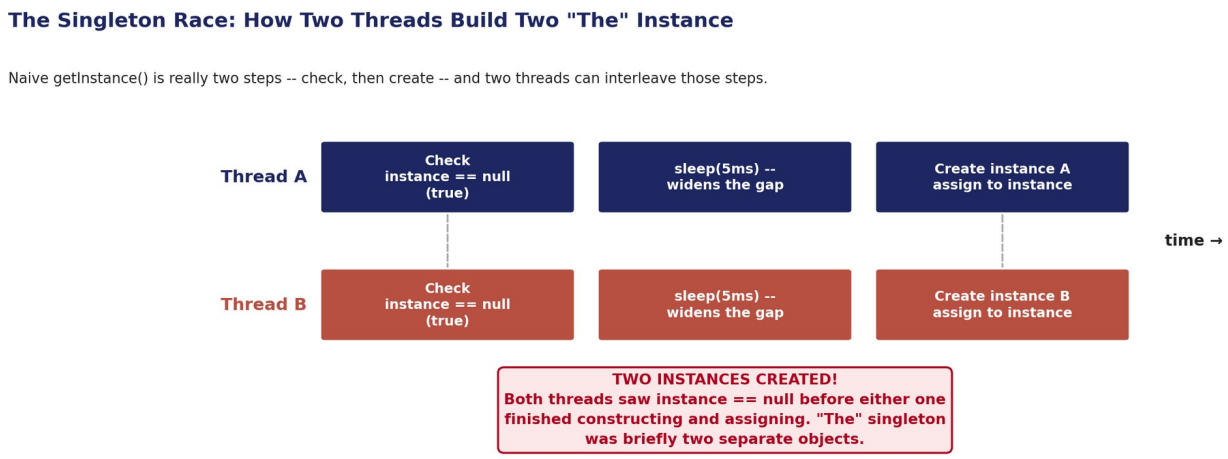


Figure 15.2 — Two threads both observe instance == null before either one finishes constructing and assigning: "the" singleton is briefly two separate objects.

Honesty note: this exact race did not appear on its own in this book's own testing

Twenty threads racing through get_instance() with an unmodified, undelayed constructor produced only ONE distinct instance, every single time across five separate runs -- even with a threading.Barrier starting gate releasing all twenty threads at once, ordinary interpreter 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), independently re-confirmed here for a different hazard shape. Per this book's standing honesty discipline, that negative finding is disclosed here rather than silently worked around: naive_singleton.py's own source carries a short time.sleep(0.005) between the check and the construction, widening a real, already-present hazard to make it observable on demand -- it does not fabricate a hazard that was not already there. With that change in place (and the threading.Barrier starting gate already present), the race reproduced reliably across three repeated runs: twenty distinct NaiveSingleton instances, every time. See Appendix 15.A for the real captured output.

safe_singleton.py -- eager initialization removes the race entirely (complete)

```
class SafeSingleton:
    _instance: "SafeSingleton"

    def __init__(self) -> None:
        pass

    @classmethod
    def get_instance(cls) -> "SafeSingleton":
        return cls._instance

SafeSingleton._instance = SafeSingleton()
```

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 get_instance() looks like from the outside.

Why eager initialization is race-free, not just less likely to race

Python's module import system guarantees a module's top-level code -- including the SafeSingleton._instance = SafeSingleton() assignment right after the class body -- runs exactly once, the first time the module is imported, before any thread outside that import could possibly call get_instance(). There is no check-then-act gap here for two threads to interleave through; this is an architectural fix, not a Lock retrofit on the naive version.

15.6 Design Patterns and Library: A Case Study

Two Patterns Applied to One Library

Factory Method centralizes WHICH class to build; Observer lets Library announce changes without knowing who is listening.

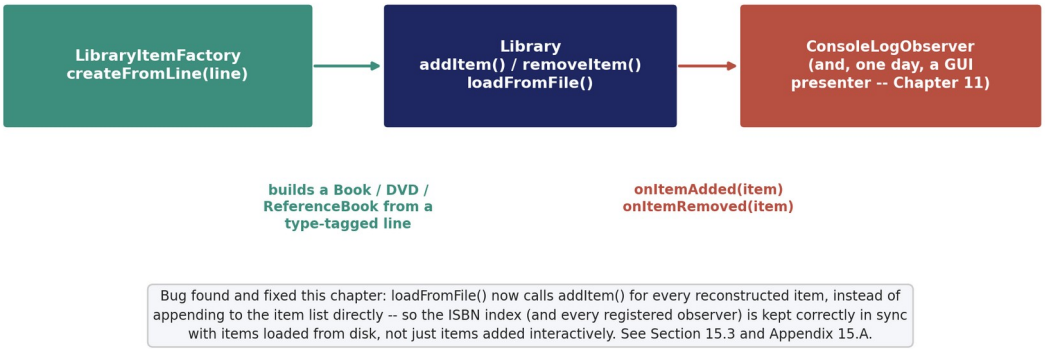


Figure 15.4 — Factory Method and Observer, both applied to the same Library this course has built since Chapter 9: library_item_factory centralizes creation, Library notifies registered observers on every change.

patterns_demo.py 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 find_by_isbn() 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 -- library_demo.py in particular -- was re-run unmodified against this chapter's new library.py 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 Running the Programs

```
$ python3 -m py_compile *.py
$ python3 patterns_demo.py
$ python3 singleton_race_demo.py
$ python3 library_demo.py
```

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 function handles this source's format" behind a single factory function, the same shape as create_from_line() -- 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 library_item_factory.create_from_line(), a pure refactor of Chapter 9/10 behavior.
- That refactor surfaced and fixed a genuine, previously undetected bug: load_from_file() never updated the _by_isbn index, silently breaking find_by_isbn() for any item ever reloaded from disk since Chapter 9/10.

- Observer lets Library announce add_item()/remove_item() 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 and NOT protected by the GIL; eager initialization removes the hazard architecturally rather than locking 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 _to_line() (serialization) stays inside Library while create_from_line() (reconstruction) moved to library_item_factory. Explain, in your own words, why these two responsibilities were NOT merged into the same module 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 whose on_item_added() raised an exception. Based on the _notify_item_added() implementation shown in Section 15.4, what would happen to the OTHER registered observers, and to add_item() 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, even with a threading.Barrier starting gate already in place. 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.py, dvd.py, item_not_found_error.py, library.py, library_demo.py, library_item.py, library_utils.py, pair.py, reference_book.py, renewable.py, library_item_factory.py, library_observer.py, console_log_observer.py, naive_singleton.py, safe_singleton.py, singleton_race_demo.py, and patterns_demo.py files (Code/Python/Chapter15/, provided alongside this book) were run on Python 3.12 before this chapter was written. Output is reproduced verbatim below from real runs.

```
$ python3 patterns_demo.py
--- Observer: notified on add_item() ---
[observer] added:   Clean Code
[observer] added:   The Imitation Game
[observer] added:   Encyclopedia of Computer Science

--- Observer: notified on remove_item() ---
[observer] removed: The Imitation Game

--- Factory Method: load_from_file() reconstructs items via library_item_factory
---
[observer] added:   Clean Code
[observer] added:   Encyclopedia of Computer Science

--- Bug found and fixed this chapter: find_by_isbn() after reload ---
reloaded.size()                          -> 2
reloaded.find_by_isbn("9780132350884")     -> Clean Code  (was None before this
chapter's refactor -- see Section 15.3)
```

Before/after verification of the find_by_isbn bug fix

A throwaway test run against the OLD (pre-Chapter-15) library.py confirmed the bug empirically: after a save/load round trip, find_by_isbn("1111111111111") returned None while find_by_title("Test Book") correctly returned the item. The identical test run against this chapter's NEW library.py returned the correct item from find_by_isbn() after the same round trip -- confirming the Factory-Method-driven rewrite of load_from_file() genuinely fixes the defect, not merely relocates it.

```
$ python3 singleton_race_demo.py
--- 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 time.sleep(0.005) widening described in Section 15.5 was 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] Python Software Foundation. "typing — Support for type hints" (Protocol, runtime_checkable). Python 3.12 documentation, accessed August 2026 (Chapter 7/13 cross-reference).

[3] Python Software Foundation. "threading — Thread-based parallelism" (Barrier, Lock, the Global Interpreter Lock). Python 3.12 documentation, accessed August 2026 (Chapter 13 cross-reference; Barrier used in this chapter's singleton_race_demo.py 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 formal Python interface or ABC. (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, lists and simple data structures, exceptions and file I/O, ABCs and Protocols, the Midterm, inheritance and polymorphism, generics-flavored typing and the collections it built on, and GUI programming under Model-View-Controller. Lecture 13 took Library beyond a single thread of execution, introducing the threading module, Lock, and the lost-update race that a naive read-modify-write sequence can produce under concurrent access — with CPython's GIL disclosed honestly as protecting individual bytecode operations, not multi-step sequences. Lecture 14 opened Library up across a network, building a client-server pair over Python's socket module, 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, the latter expressed as a Protocol — 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.

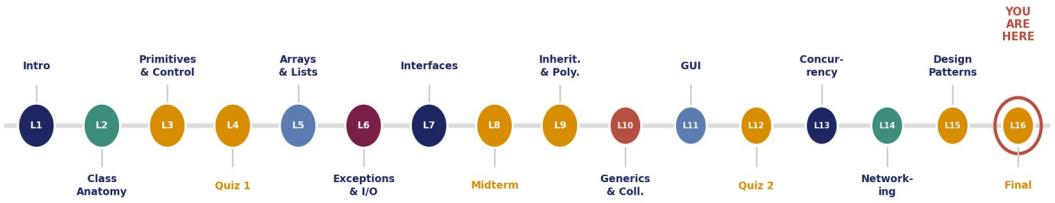


Figure 16.1 — This chapter sits at Lecture 16, the Final, at the end of the sixteen-lecture OOP roadmap: a capstone, not a new topic.

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 Tkinter/PyQt components
2	Concurrent Server	L13 -- threading, locks, avoiding lost-update races
3	Client-Server Protocol	L14 -- the socket module, 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 formal Python Protocol or ABC. 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 docstrings, 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 (including which packages `pip install` must fetch), 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 ConnectionError. 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 the Library client and server with explicit try/except around every socket operation, specifically because a network call can fail in ways a local function 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 docstring on a function 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] Python Software Foundation, "socket — Low-level networking interface," Python 3.12 documentation, accessed August 2026.

[3] Python Software Foundation, "threading — Thread-based parallelism," Python 3.12 documentation, accessed August 2026.