OBJECT-ORIENTED PROGRAMMING WITH JAVA

Java™

A Practical Guide to Software Design and Application Development

A TEXTBOOK FOR
COMPUTER SCIENCE STUDENTS
AND FUTURE-READY DEVELOPERS

JEFFREY V. TOLENTINO

OBJECT-ORIENTED PROGRAMMING WITH JAVA

A Practical Guide to Software Design
and Application Development

Jeffrey V. Tolentino

Copyright Page

Object-Oriented Programming with Java: A Practical Guide to Software Design and Application Development

Published by: Philippine Technical Institute Inc.
Guilig Street, Poblacion, Lingayen, Pangasinan 2401, Philippines

Mobile No.: +63 918 620 0902
Email: po@pti.edu.ph
Website: press.pti.edu.ph

Date of Publication: June 30, 2026

ISBN:
PDF Downloadable: 978-621-8546-10-3
PDF Read-Only: 978-621-8546-09-7

Cover Design: PTI Press Creative Team

Published in the Philippines by PTI Press.

Foreword

Few skills have proven as durable in computing education as the ability to think in objects. Programming languages rise and fall in popularity, frameworks are born and retired, and the tools of the trade are remade with each passing decade, yet the discipline of modeling a problem as a set of well-designed objects, each bundling its own data with the behavior that acts on it, has remained a stable foundation of professional software development for more than a generation. A textbook that teaches this discipline well, through a language as widely used and well-suited to teaching it as Java, gives students something that will serve them long after any particular tool has been superseded.

This book is notable for the deliberate way it builds that understanding. Rather than presenting object-oriented programming as a collection of features to be memorized, it develops it as a way of thinking, guiding the student from the first simple program through the pillars of encapsulation, inheritance, and polymorphism to the design of a complete application. Its cumulative Application Build, carried across all fourteen chapters, ensures that the concepts are not merely studied but assembled into working software, which is the truest test of whether they have been learned.

For students in Philippine computer science programs preparing to enter a software industry that increasingly demands genuine object-oriented design competency, this text offers exactly the disciplined, practical foundation the field requires. It is offered in that spirit: as a foundation to build upon, and a beginning rather than an end.

Mr. Marvin Santillian

Preface

This book grew from a simple observation: students learning Java often master the language's syntax before developing the object-oriented way of thinking that gives it purpose. They may learn to write classes without fully understanding why classes are useful for modeling problems, or encounter inheritance and polymorphism as isolated features rather than parts of a unified design philosophy. This book makes those connections clear, guiding students from their first Java program to the development of a complete object-oriented application.

The book follows the SDF104 study guide and the Java object-oriented programming syllabus. Its lessons are organized into 14 chapters across 5 Parts, with each topic deliberately building on the previous one. Every code example has been compiled and tested, and its verified output is provided so that students and instructors can use the examples with confidence.

This edition follows the Philippine Technical Institute Inc. (PTI Press) house style. Before final publication, the references will undergo a complete APA 7th edition accuracy audit, and the content will be checked against the final approved syllabus for any required adjustments.

At the center of the book is the Application Build, a cumulative project developed throughout the fourteen chapters. Students begin by selecting a realistic small application and gradually improve it as new concepts are introduced. Data types become fields, classes are created, encapsulation protects data, inheritance and polymorphism organize behavior, and exception handling strengthens reliability. By Chapter 14, students will have completed a functional object-oriented application.

For instructors, each Part serves as a natural instructional unit: Java Foundations, Control and Data Structures, Classes and Objects, the Pillars of Object-Oriented Programming, and Application Development. The Application Build may serve as the course's primary summative assessment, while the Appendix A worksheet packet may be distributed at the beginning of the term as a continuing companion resource.

Jeffrey V. Tolentino

How to Use This Book

Each chapter follows a consistent structure designed to support both study and teaching. A chapter opens with an Overview and Learning Outcomes, then introduces its topic through an Introduction and Historical Background that give context before details. A Definitions section gathers key terms, and a Core Concepts section develops the material with a worked, verified code example at its center. Each chapter then offers a summary table, a discussion of standards and the Philippine context with an illustrative case study, best practices and common pitfalls, and a Summary with Key Takeaways. Every chapter closes with Key Terms, reflection and discussion prompts, a step of the cumulative Application Build, and a self-check.

Students are encouraged to type and run each code example rather than merely read it, since the verified output noted in each caption gives an immediate check that the program behaves as intended. The Application Build activity at the end of each chapter should be treated as an ongoing project carried across the whole book, not as isolated exercises; its steps are cumulative, and completing them in sequence produces a finished application by the final chapter.

Acknowledgments

This book builds directly on the SDF104 Java Programming Fundamentals study guide and the Java object-oriented programming course syllabus (Pangasinan State University), whose topic sequence, from data types and control structures through classes, the object-oriented pillars, and application development, provided the foundational structure this fourteen-chapter, five-Part expansion restructures and deepens. The book's development process mapped each chapter's scope back to the syllabus's three course outcomes: understanding and designing object-oriented concepts, applying standards and principles to write readable object-oriented code, and designing and developing a computing solution using an object-oriented approach.

Thanks are due to the faculty and students of the computer science program whose questions and difficulties shaped the explanations offered here, and to the many authors of the foundational Java and object-oriented design literature, cited throughout, on whose work any introductory treatment necessarily stands.

Table of Contents

PART I

JAVA FOUNDATIONS

Part I establishes the groundwork for everything that follows. Before objects and classes can make sense, the student must understand what Java is and why it is object-oriented, the primitive data and operators a program manipulates, and how a program reads input and produces output to become interactive.

INTRODUCTION TO JAVA AND OBJECT-ORIENTED THINKING

Why Java, and Why Objects

"Programming is not about instructing a machine step by step. It is about modeling a problem so clearly that the solution becomes obvious."

1.0 Chapter Overview

This chapter opens your study of object-oriented programming by introducing the Java language, the platform it runs on, and the way of thinking that distinguishes object-oriented programming from the procedural style you may have met first. Before you write your first class or method, you need to understand what Java is, why it became one of the most widely used languages in the world, and what it means to model a problem in terms of objects. This chapter also introduces the Application Build, the cumulative project you will develop across all fourteen chapters, growing a single working Java application one concept at a time.

1.1 Learning Outcomes

- LO 1.1 Explain what Java is, how it runs, and why it is widely used.
- LO 1.2 Distinguish object-oriented programming from procedural programming.
- LO 1.3 Write, compile, and run a first simple Java program.

1.2 Introduction

Java is a general-purpose, object-oriented programming language designed to be portable, reliable, and approachable. Since its release in the mid-1990s, it has become one of the most widely used languages in the world, powering everything from enterprise business systems and Android applications to embedded devices and large-scale web services. For a student learning to program, Java offers an unusually good combination of qualities: it enforces a disciplined,

object-oriented structure that teaches good habits, while remaining readable enough that beginners can follow what a program does.

This course is specifically about object-oriented programming, and that phrase deserves attention from the outset. Object-oriented programming is not merely a set of Java features to memorize; it is a way of thinking about problems, in which a program is organized around objects that bundle data and the behavior that acts on it. Learning to think this way is the central goal of this entire book, and every chapter builds toward the point where you can design and develop a complete application using object-oriented principles rather than merely writing lines of code.

1.3 Historical Background

Java was developed at Sun Microsystems in the early 1990s, originally for interactive television and embedded devices, and was publicly released in 1995. Its defining design goal was captured in the slogan write once, run anywhere: a Java program is compiled not to the machine code of any specific processor, but to an intermediate form called bytecode, which runs on the Java Virtual Machine (JVM). Because a JVM exists for virtually every platform, the same compiled Java program runs unchanged across Windows, Linux, macOS, and many other systems, a portability that was revolutionary at the time and remains central to Java's appeal.

Object-oriented programming itself predates Java by decades, with roots in languages such as Simula and Smalltalk, but Java brought object-oriented design into the mainstream of professional software development on an enormous scale. By making classes and objects the mandatory organizing structure of every program, Java ensured that a generation of developers learned to think in objects, and the concepts this book teaches, encapsulation, inheritance, and polymorphism, became the shared vocabulary of modern software engineering far beyond Java itself.

1.4 Definitions

KEY TERMS

Java: A general-purpose, object-oriented programming language that compiles to platform-independent bytecode.

Bytecode: The intermediate, platform-independent form of a Java program is compiled to, run by the Java Virtual Machine.

Java Virtual Machine (JVM): The software that executes Java bytecode on a specific platform, enabling write once, run anywhere.

Object-Oriented Programming (OOP): A programming paradigm that organizes a program around objects, which bundle data and the behavior acting on it.

JDK: The Java Development Kit, the toolset (including the compiler) used to write and build Java programs.

1.5 Core Concepts of Object-Oriented Thinking

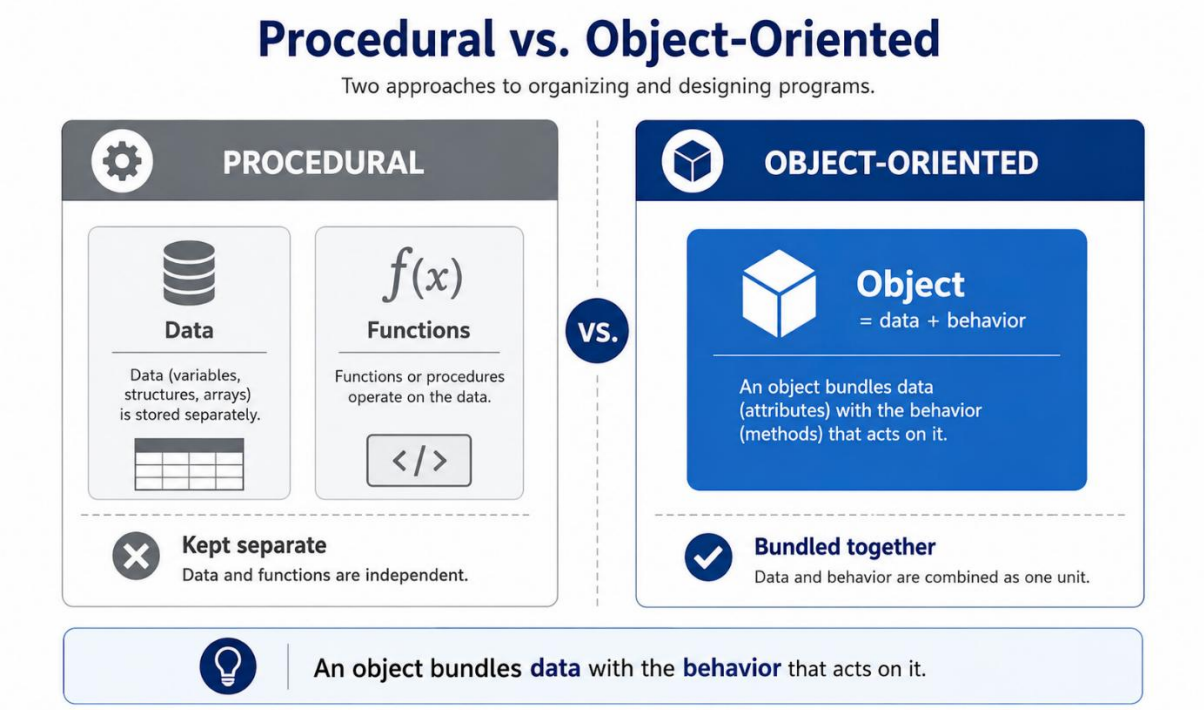

Figure 1.1. Procedural programming keeps data and behavior separate; object-oriented programming bundles them into objects.

1.5.1 Procedural vs. Object-Oriented

In the procedural style, a program is organized around procedures, or functions, that operate on data passed to them; the data and the operations on it are kept separate. In the object-

oriented style, data and the operations that act on it are bundled together into objects. Consider modeling a student: procedurally, you might keep separate variables for the name, identification number, and grade, and write standalone functions to manipulate them. In the object-oriented style, you create a Student object that holds those values and knows how to act on them, keeping the data and behavior together as a single, coherent unit.

1.5.2 Classes and Objects

The two most fundamental ideas in object-oriented programming are the class and the object. A class is a blueprint or template that defines what data an object will hold and what it can do; an object is a specific instance created from that class. The relationship is like that between an architectural blueprint and the houses built from it: one blueprint (the class) can produce many individual houses (the objects), each with its own specific characteristics. This distinction between the general template and the specific instance is the foundation of everything else in this book, and Chapters 7 through 9 develop it in full detail.

1.5.3 Your First Java Program

Every Java program is built from classes, and even the simplest program must be enclosed in one. The traditional first program prints a greeting to the screen. In the listing below, the class is named Hello, and the special method main is the entry point where the program begins running when it is launched.

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

Listing 1.1. A first Java program, verified to print: Hello, world!

Several elements in this small program will recur in every Java program you write. The keyword public class Hello declares a class named Hello. Inside it, public static void main(String[] args) declares the main method, the fixed entry point that the Java launcher looks for. Within main, the statement System.out.println prints a line of text to the console. Do not worry about understanding every keyword yet; the following chapters explain each in turn. What matters now is recognizing that even this trivial program is organized as a class, because in Java, everything lives inside a class.

1.6 Table 1.1, Procedural vs. Object-Oriented

Aspect	Procedural	Object-Oriented
Organized around	Functions/procedures	Objects
Data and behavior	Kept separate	Bundled together
Core unit	The function	The class and its objects
Models the world as	Steps to perform	Things that have state and behavior

Table 1.1. Two ways of organizing a program.

1.7 The Java Development Workflow

Writing and running a Java program involves three conceptual steps. First, write the source code in a plain-text file with a .java extension, matching the class name (so the Hello class lives in Hello.java). Second, the Java compiler translates this source into bytecode. Third, the Java Virtual Machine executes that bytecode. In professional practice, an integrated development environment (IDE) such as IntelliJ IDEA, Eclipse, or NetBeans automates these steps and adds tools for editing, debugging, and organizing code, but understanding the underlying compile-and-run cycle helps you reason about what is actually happening when your program runs.

1.8 Philippine Context and Case Study

PHILIPPINE CONTEXT

Java remains one of the most in-demand programming languages in the Philippine information technology industry, particularly across the country's large business process outsourcing, financial technology, and enterprise software sectors, and Android mobile development keeps Java-family skills relevant to the mobile-first Philippine market. For a computer science student, solid object-oriented Java competency is among the most directly employable skills this curriculum can build, which is precisely why this course grounds its teaching in careful, disciplined object-oriented practice rather than surface familiarity.

CASE STUDY

Illustrative composite case.

A Philippine software team maintaining a growing enterprise system found that its early code, written in a procedural style with data and logic scattered across many functions, had become increasingly difficult to modify safely, since a change in one place tended to break something unexpected elsewhere. Restructuring the system around well-defined objects, each bundling its own data with the behavior that acted on it, made the codebase far easier to reason about and change, because each object's responsibilities were clearly contained. The team's experience illustrates the practical payoff of object-oriented thinking that this book develops: not cleverness for its own sake, but code that remains understandable and safe to change as it grows.

1.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Approach object-oriented programming as a way of thinking about problems, not merely a set of syntax rules to memorize.
- Best Practice: Name each source file to match its public class exactly, since Java requires this correspondence.
- Emerging Trend: Modern Java continues to evolve rapidly, adding features that make it more concise, but the object-oriented fundamentals this book teaches remain its stable core.
- Common Pitfall: Writing procedural-style code inside Java classes, keeping data and behavior separate rather than bundling them into objects.
- Common Pitfall: Confusing a class (the template) with an object (a specific instance created from it).

- Common Pitfall: Mismatching the file name and the public class name, which prevents the program from compiling.

1.10 Summary and Key Takeaways

Java is a portable, object-oriented language that compiles to bytecode run by the Java Virtual Machine, giving it its write once, run anywhere portability. Object-oriented programming organizes a program around objects that bundle data and behavior, in contrast to the procedural style, which keeps them separate. The class is a blueprint and the object a specific instance created from it, a distinction at the heart of everything that follows. Chapter 2 now turns to the data types, variables, and operators that programs manipulate.

KEY TAKEAWAYS

- Java compiles to platform-independent bytecode run by the JVM, enabling write once, run anywhere.
- Object-oriented programming bundles data and behavior into objects; procedural programming keeps them separate.
- A class is a blueprint; an object is a specific instance created from it.
- Every Java program is organized as one or more classes, with main as the entry point.

1.11 Key Terms, Reflection, and Discussion

Java. Bytecode. Java Virtual Machine. Object-Oriented Programming. Procedural Programming. Class. Object. JDK. main method.

- Reflection: Think of an everyday thing, a car, a bank account, a library book. What data would it hold, and what behavior would it have, if modeled as an object?

- Discussion: Java's portability came from compiling to bytecode rather than native machine code. What might such portability cost in exchange, and is the trade worthwhile?

1.12 Activity and Application Build, Step 1

APPLICATION BUILD, STEP 1

Decide on the application you will build across this entire book (a realistic small application: a student records manager, a simple banking system, an inventory tracker, or a library catalog). Write a one-paragraph description of what it does and who uses it. Then list three or four real-world things it deals with, students, accounts, items, that will become the objects your program models in later chapters.

1.13 References

- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
- Schildt, H. (2021). Java: The complete reference (12th ed.). McGraw-Hill.
- Oracle. The Java tutorials. Oracle Corporation.

End of Chapter 1. Continue to Chapter 2: Data Types, Variables, and Operators

DATA TYPES, VARIABLES, AND OPERATORS

The Raw Materials a Program Works With

"Before a program can reason about the world, it must first be able to hold a piece of it: a number, a character, a truth value."

2.0 Chapter Overview

Every program works with data, and this chapter introduces how Java represents and manipulates the most basic kinds of it. You will learn Java's primitive data types, how to declare variables to hold values, how to name them with identifiers, and how operators combine and transform them. This material is the raw vocabulary of programming; every later chapter, however advanced, ultimately rests on the ability to store a value in a variable and compute with it.

2.1 Learning Outcomes

- LO 2.1 Differentiate among Java literals, primitive data types, variables, identifiers, and operators.
- LO 2.2 Declare and initialize variables of the appropriate type.
- LO 2.3 Use arithmetic, relational, and logical operators, respecting operator precedence.

2.2 Introduction

A program manipulates data, and Java requires you to be explicit about what kind of data each piece is. A whole number, a number with a fractional part, a single character, and a true-or-false value are all represented differently, and Java's type system tracks these differences to help catch errors and use memory efficiently. This chapter introduces the primitive data types that represent these basic values, the variables that hold them, and the operators that act on them.

Although this material may feel elementary compared to the objects and classes that lie ahead, mastering it is essential because these primitive values and the operations on them are the substance that objects ultimately manage. A Student object, when you build one in a later chapter, will hold an identification number (an integer), a grade (a floating-point number), and a name (a sequence of characters), and it will compute with them using exactly the operators introduced here.

2.3 Historical Background

Java's system of primitive types, with fixed, well-defined sizes for each type regardless of platform, was a deliberate design choice inherited from its portability goal. In many earlier languages, the size of a basic type such as an integer could vary from one machine to another, causing subtle bugs when programs moved between platforms. Java fixed the size and behavior of each primitive type as part of the language specification, so that an int is exactly the same everywhere, reinforcing the write once, run anywhere promise at the most basic level of data.

This disciplined, explicit approach to types reflects a broader philosophy in Java's design: catch as many errors as possible early, during compilation, rather than late, when the program runs and fails in front of a user. Requiring the programmer to declare the type of every variable lets the compiler verify that operations make sense, a form of safety that has made Java well-suited to large, long-lived software systems where reliability matters more than brevity.

2.4 Definitions

KEY TERMS

Primitive Data Type: One of Java's basic built-in types, such as int, double, char, and boolean, that holds a simple value directly.

Variable: A named storage location that holds a value of a declared type.

Literal: A fixed value written directly in code, such as 42, 3.14, 'A', or true.

Identifier: The name given to a variable, method, or class, following Java's naming rules.

Operator: A symbol that performs an operation on values, such as + for addition or > for comparison.

2.5 Core Concepts of Data and Operators

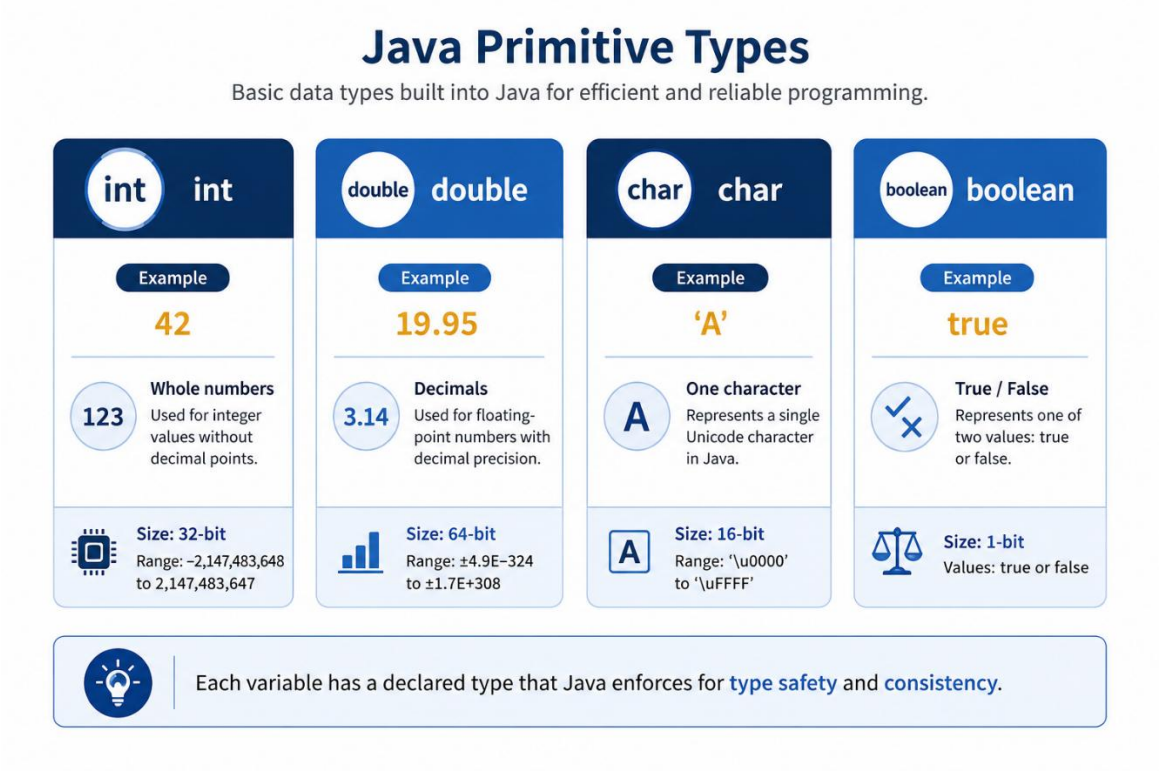

Figure 2.1. Java's four most commonly used primitive types, each holding a distinct kind of value.

2.5.1 Primitive Types and Variables

Java provides several primitive data types, of which four are used constantly: int holds whole numbers, double holds numbers with a fractional part, char holds a single character, and boolean holds a true-or-false value. To use a value, you declare a variable of the appropriate type and assign it a value. A variable declaration states the type, then the identifier (the name), and usually an initial value. The identifier must follow Java's naming rules: it may contain letters, digits, and underscores, must not begin with a digit, and by convention begins with a lowercase letter for variables.

2.5.2 Operators

Operators act on values to produce new ones. The arithmetic operators (+, -, *, /, and % for remainder) perform calculations. The relational operators (such as >, <, ==, and !=) compare values and produce a boolean result. The logical operators (&& for and, || for or, ! for not) combine boolean values. The assignment operator (=) stores a value in a variable. A crucial subtlety appears with division: dividing two integers performs integer division, discarding any

fraction, so 7 / 2 yields 3, not 3.5; to get a fractional result you must involve a floating-point value, as in 7 / 2.0.

2.5.3 A Worked Example

The program below declares variables of several primitive types, combines a string with other values, and demonstrates the integer-versus-floating-point division subtlety that trips up many beginners.

```
public class Types {
    public static void main(String[] args) {
        int count = 42;
        double price = 19.95;
        char grade = 'A';
        boolean passed = true;
        String message = "Total items: ";
        System.out.println(message + count);
        System.out.println("Price: " + price + ", Grade: " + grade + ", Passed: " + pass
ed);
        int sum = count + 8;
        System.out.println("Sum is " + sum + ", average is " + (sum / 2.0));
    }
}
```

Listing 2.1. Declaring variables and using operators. Verified output: Total items: 42 / Price: 19.95, Grade: A, Passed: true / Sum is 50, average is 25.0

Notice how the + operator does two different things here: between two numbers it adds, but when one side is a String it joins, or concatenates, the values into a single string of text. Notice too that sum / 2.0 produces 25.0 rather than 25, because dividing by the floating-point 2.0 rather than the integer 2 gives a floating-point result. These two behaviors, string concatenation with + and the integer-versus-floating-point division rule, are among the most common sources of early confusion, and recognizing them now will save considerable puzzlement later.

2.6 Table 2.1, Common Primitive Types

Type	Holds	Example Literal
int	Whole numbers	42
double	Numbers with a fractional part	19.95
char	A single character	'A'

Type	Holds	Example Literal
boolean	A true or false value	true

Table 2.1. Four commonly used primitive types.

2.7 Operator Precedence

When an expression contains several operators, Java evaluates them in a defined order of precedence, much like the order of operations in arithmetic. Multiplication, division, and remainder are evaluated before addition and subtraction, so 2 + 3 * 4 yields 14, not 20. When in doubt, or to make intent unmistakable, use parentheses to force the order you want; the expression (2 + 3) * 4 yields 20. Relational and logical operators have lower precedence and are evaluated after arithmetic. Rather than memorizing the full precedence table, the practical rule is simple: when an expression mixes different operators, add parentheses to make the intended order explicit, both for correctness and for the benefit of anyone reading the code later.

2.8 Philippine Context and Case Study

PHILIPPINE CONTEXT

Careful attention to data types matters in real Philippine software, particularly in the financial and payroll systems common in the country's outsourcing and enterprise sectors, where using a floating-point type for currency carelessly, or mishandling integer division, can produce small but unacceptable monetary errors. A developer who understands from the outset when integer division truncates and how numeric types behave writes code that avoids an entire category of subtle, costly bugs, a discipline this chapter aims to instill early.

CASE STUDY

Illustrative composite case.

A student building a grade-averaging feature computed an average by dividing the sum of several integer scores by the number of scores, and was baffled when a set of

scores that clearly averaged around 85.5 reported an average of exactly 85. The cause was integer division: dividing one integer by another discarded the fractional part. Changing the calculation to divide by a floating-point value, or to store the total as a double, produced the correct 85.5, illustrating exactly the integer-versus-floating-point division subtlety this chapter emphasizes, and why understanding types is a practical necessity rather than a formality.

2.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Choose the type that fits the data, int for whole quantities, double for measurements, and be deliberate about currency.
- Best Practice: Use clear, descriptive identifier names beginning with a lowercase letter for variables, following Java convention.
- Best Practice: Add parentheses to make the intended order of operations explicit in any mixed expression.
- Common Pitfall: Dividing two integers and losing the fractional part, when a floating-point result was intended.
- Common Pitfall: Confusing the assignment operator (=) with the equality comparison operator (==).
- Common Pitfall: Assuming + always adds, when between a string and another value it concatenates instead.

2.10 Summary and Key Takeaways

Java represents basic data with primitive types, int, double, char, and boolean chief among them, held in variables named by identifiers and manipulated by operators. Arithmetic, relational, and logical operators combine values, following a defined precedence that parentheses can make explicit. Two behaviors deserve early attention: the + operator concatenates when a string is involved, and dividing two integers truncates the fraction. Chapter 3 now turns to getting data into and out of a program.

KEY TAKEAWAYS

- Primitive types (int, double, char, boolean) hold basic values; variables named by identifiers store them.
- Operators perform arithmetic, comparison, and logic, following a defined precedence.
- The + operator concatenates when a string is involved, and adds when both sides are numbers.
- Dividing two integers truncates the fraction; involve a floating-point value for a fractional result.

2.11 Key Terms, Reflection, and Discussion

Primitive Data Type. Variable. Literal. Identifier. Operator. int. double. char. boolean. Operator Precedence. Concatenation.

- Reflection: For the objects your Application Build models, what data does each hold, and which primitive type fits each piece?

- Discussion: Java requires you to declare the type of every variable, while some languages infer it automatically. What are the advantages and costs of Java's explicit approach?

2.12 Activity and Application Build, Step 2

APPLICATION BUILD, STEP 2

For each object your application models (from Step 1), list the pieces of data it must hold and assign each an appropriate Java primitive type (or String). For example, a student might hold a name (String), an identification number (int), and a grade (double). This data inventory becomes the fields of your classes in later chapters.

2.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. Which primitive type holds a number with a fractional part?

a) int b) double c) char d) boolean

2. In Java, the expression 7 / 2 evaluates to:

a) 3.5 b) 3 c) 4 d) 3.0

3. When one operand of + is a String, the operator:

a) Adds numerically b) Concatenates into a string c) Causes an error d) Compares the values

4. Which is a relational operator?

a) + b) = c) > d) &&

5. To make the intended order of operations explicit in a mixed expression, use:

a) More variables b) Parentheses c) A different type d) A semicolon

2.14 References

- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
- Schildt, H. (2021). Java: The complete reference (12th ed.). McGraw-Hill.
- Oracle. The Java tutorials: Language basics. Oracle Corporation.

End of Chapter 2. Continue to Chapter 3: Getting Input and Producing Output

GETTING INPUT AND PRODUCING OUTPUT

Making a Program Interactive

"A program that cannot receive input can only ever tell you what you already knew when you wrote it."

3.0 Chapter Overview

This chapter closes Part I by making programs interactive: able to receive user input and produce output in response. You will learn the standard ways Java reads input from the keyboard, principally the Scanner class, along with the BufferedReader and JOptionPane alternatives, and how to convert text input into the numeric types you learned in Chapter 2. An interactive program is the first that feels genuinely useful, and input and output are the foundation of every application you will build hereafter.

3.1 Learning Outcomes

- LO 3.1 Produce output to the console using System.out.
- LO 3.2 Read input from the keyboard using the Scanner class.
- LO 3.3 Recognize the BufferedReader and JOptionPane alternatives and the need to parse text into numbers.

3.2 Introduction

The programs in the previous chapters produced output but took no input; they did exactly the same thing every time they ran. A genuinely useful program responds to its user, reading data provided at runtime and producing results that depend on it. This chapter introduces the standard ways a Java program reads input from the keyboard and produces output to the console, turning a static program into an interactive one.

Java offers several mechanisms for reading input. The Scanner class is the most convenient and widely taught, and this book uses it throughout. The older BufferedReader approach and

the graphical JOptionPane dialog offer alternatives suited to particular situations. All of them share a common concern that this chapter highlights: input arrives as text, and converting that text into the numeric types your program computes with, a step called parsing, is a routine but essential part of handling input correctly.

3.3 Historical Background

Early Java input relied on the BufferedReader class wrapped around the standard input stream, an approach that worked but required several lines of setup and explicit conversion of text to numbers. Recognizing that this was cumbersome for beginners and common cases, the Java platform introduced the Scanner class in 2004, which reads and converts input in a single, readable step. This addition significantly lowered the barrier to writing interactive programs and is why Scanner has become the standard tool taught in introductory Java courses today.

The JOptionPane class, part of Java's graphical user interface toolkit, offers yet another approach: pop-up dialog boxes for input and output. It reflects a period when desktop graphical applications were central, and while console input via Scanner dominates in learning and server-side software, graphical input remains relevant. Chapter 14 returns to graphical interfaces when the book turns to full application development. The coexistence of these approaches reflects the different contexts in which Java programs run: console, stream-based, and graphical.

3.4 Definitions

KEY TERMS

Scanner: A Java class that reads and converts input from a source such as the keyboard in a convenient, readable way.

BufferedReader: An older Java class for reading text input from a stream, requiring explicit conversion of text to other types.

JOptionPane: A class that provides simple graphical dialog boxes for input and output.

Parsing: Converting text into another type, such as turning the string "42" into the integer 42.

Standard Input / Output: The default keyboard input (System.in) and console output (System.out) streams.

3.5 Core Concepts of Input and Output

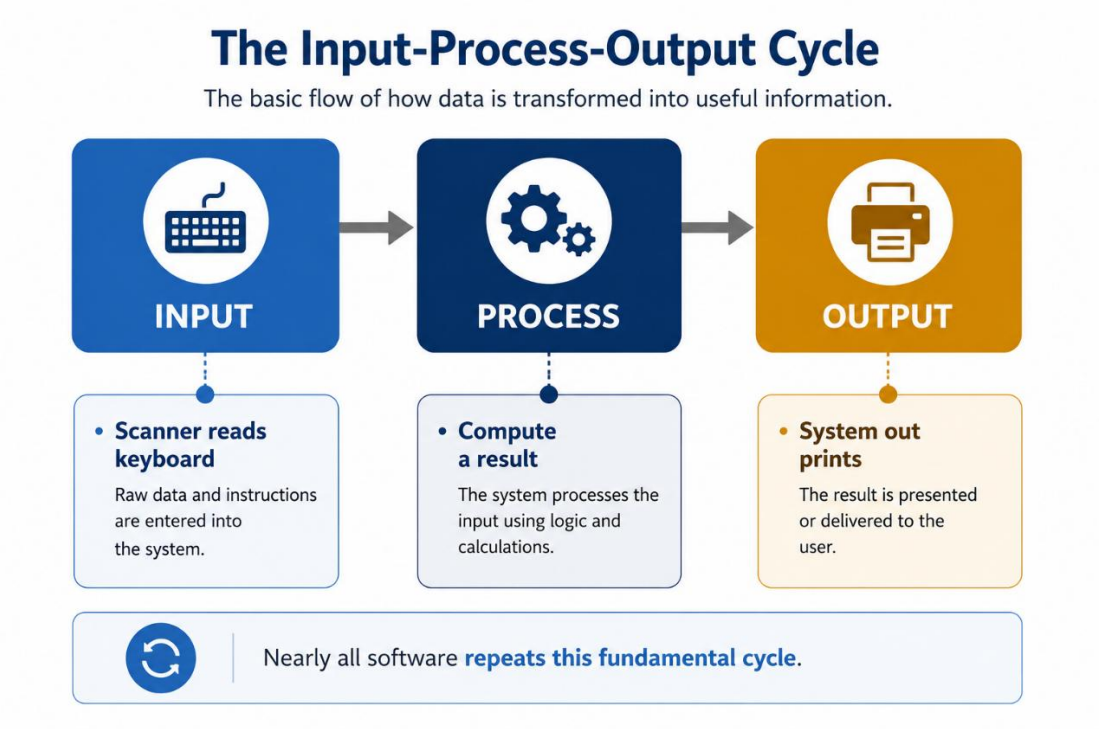

Figure 3.1. The input-process-output cycle that underlies nearly all software.

3.5.1 Producing Output

Output to the console is produced with System.out. The method println prints its argument followed by a new line, while print prints without moving to a new line, letting the next output continue on the same line, which is useful for prompts. You have already seen println in the previous chapters; combined with string concatenation, it is sufficient for all the console output this book requires until Chapter 14.

3.5.2 Reading Input with Scanner

To read input with Scanner, you first create a Scanner connected to the standard input stream System.in, then call methods on it to read values: nextLine reads a whole line of text, nextInt reads an integer, and nextDouble reads a floating-point number. The Scanner automatically converts typed text to the requested type, which is a major part of its convenience. The program below prompts the user for a name and an age, reads both, and uses them in its output.

```
import java.util.Scanner;

public class InputDemo {
```

```
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = input.nextLine();
        System.out.print("Enter your age: ");
        int age = input.nextInt();
        System.out.println("Hello, " + name + ". Next year you will be " + (age + 1) + "
.");
    }
}
```

Listing 3.1. Reading input with Scanner. Verified: given "Maria" and 20, prints "Hello, Maria. Next year you will be 21."

The line import java.util.Scanner at the top makes the Scanner class available; import statements, examined further in a later chapter, tell Java where to find classes that are not part of the always-available core. Notice that nextInt reads the age as an integer directly, so the expression age + 1 performs genuine numeric addition. Had the age been read as text, it would have needed parsing into an integer first, the conversion step the next subsection addresses.

3.5.3 Alternatives and Parsing

The BufferedReader approach reads input as text and requires you to convert, or parse, that text into numbers explicitly, using methods such as Integer.parseInt, which turns the string "42" into the integer 42. The JOptionPane class shows a graphical dialog box that returns the user's input as text, again requiring parsing before numeric use. Even with Scanner, parsing lurks whenever input that looks numeric is read as text, so understanding that text and numbers are distinct, and that converting between them is a deliberate step, is essential to handling input correctly regardless of which mechanism you use.

3.6 Table 3.1, Ways to Read Input

Mechanism	Style	Notes
Scanner	Console, convenient	Reads and converts in one step; used throughout this book
BufferedReader	Console, stream-based	Older; reads text, requires explicit parsing
JOptionPane	Graphical dialog	Pop-up input box; returns text, requires parsing

Table 3.1. Three ways to read keyboard input in Java.

3.7 Combining Input, Computation, and Output

With input and output in hand, you can now write a program that follows the fundamental pattern of nearly all software: read input, compute a result, produce output. This input-process-output cycle underlies applications of every size, from a simple calculator to a large enterprise system, which merely repeat and elaborate the same basic loop. As you build your application across this book, this cycle will recur constantly: the application reads what the user wants, processes it using the objects and logic you design, and reports the result. Everything else this book teaches, decisions, loops, classes, inheritance, serves to make the process step richer and better organized.

3.8 Philippine Context and Case Study

PHILIPPINE CONTEXT

Robust input handling is a practical necessity in Philippine software serving diverse users, since real users enter unexpected data, letters where numbers are expected, blank fields, mistyped values, and a program that assumes perfectly formed input will fail in the field. While full input validation and error handling await Chapter 13, an early awareness that input is text arriving from an unpredictable human, and must be converted and checked deliberately, is a habit that distinguishes reliable software from fragile prototypes.

CASE STUDY

Illustrative composite case.

A student's first interactive program read a menu choice as a number with nextInt, and worked perfectly in testing, until a classmate typed a word instead of a number and the program crashed abruptly. The experience taught the student a lesson this book returns to in Chapter 13: input from a real user cannot be assumed to be well formed, and the difference between a program that works in a demonstration and one that survives real use lies largely in how carefully it handles input that does not match

> expectations. For now, the takeaway is simply to recognize that input is unpredictable text until proven otherwise.

3.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Use Scanner for console input in learning and most straightforward cases, for its readability and built-in conversion.
- Best Practice: Print a clear prompt with print before reading input, so the user knows what to enter.
- Best Practice: Remember that input is fundamentally text; treat conversion to numbers as a deliberate step.
- Common Pitfall: Forgetting the import statement for Scanner, so the program will not compile.
- Common Pitfall: Mixing nextInt and nextLine without accounting for the leftover newline, a classic Scanner surprise.
- Common Pitfall: Assuming numeric-looking input is already a number, when it may be text that requires parsing.

3.10 Summary and Key Takeaways

Interactive programs read input and produce output. Output uses System.out with print and println; input most conveniently uses the Scanner class, with BufferedReader and JOptionPane as alternatives. Input arrives as text, and converting it to numbers, called parsing, is an essential step, implemented with specific mechanisms and built into Scanner's methods. The read-

process-output cycle, which enables applications of every size. This chapter closes Part I; Part II now turns to controlling the flow of a program with decisions, loops, and arrays.

KEY TAKEAWAYS

- Output uses System.out.print and println; input most conveniently uses the Scanner class.
- Scanner reads and converts input in one step; nextLine, nextInt, and nextDouble read different types.
- Input arrives as text; converting it to numbers (parsing) is a deliberate, essential step.
- The read-process-output cycle underlies applications of every size.

3.11 Key Terms, Reflection, and Discussion

Scanner. BufferedReader. JOptionPane. Parsing. Standard Input. Standard Output. println. nextInt. import.

- Reflection: What input will your Application Build need from its user, and what output will it produce in response?
- Discussion: Scanner made interactive Java far easier than the older BufferedReader approach. When might the more verbose BufferedReader still be preferable?

3.12 Activity and Application Build, Step 3

APPLICATION BUILD, STEP 3

Write a small interactive program that uses Scanner to gather the initial data your application needs, for example, prompting for a new student's name, identification number, and grade, and printing them back in a formatted confirmation message. This exercises the read-process-output cycle and produces input handling you will reuse when your classes take shape in later chapters.

3.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. Which class is the convenient, widely taught way to read console input in Java?

a) BufferedReader b) Scanner c) JOptionPane d) System.in

2. The method that prints text followed by a new line is:

a) print b) println c) nextLine d) read

3. Converting the text "42" into the integer 42 is called:

a) Casting b) Parsing c) Rounding d) Concatenating

4. Which Scanner method reads an integer?

a) nextLine b) nextInt c) nextText d) readInt

5. JOptionPane provides input via:

a) A console prompt b) A graphical dialog box c) A file d) A network stream

3.14 References

- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
- Oracle. The Java tutorials: Basic I/O and Scanner. Oracle Corporation.
- Schildt, H. (2021). Java: The complete reference (12th ed.). McGraw-Hill.

End of Chapter 3. Continue to Chapter 4: Decision Control Structures

PART II

CONTROL AND DATA STRUCTURES

Part II gives programs the ability to choose and to repeat. Decision structures let a program respond differently to different situations, loops let it process arbitrarily much data with concise code, and arrays let it hold and process collections of values, the first data structure and a bridge toward objects.

DECISION CONTROL STRUCTURES

Teaching a Program to Choose

"A program without decisions is a recipe. A program with them is a mind, however simple, weighing a condition and choosing a path."

4.0 Chapter Overview

Part I built programs that ran straight through from top to bottom. Part II introduces control flow, the ability of a program to choose different paths and repeat actions. This chapter covers decision-making: the if and if-else structures that let a program take different actions depending on a condition, and the related switch and conditional operator. Decisions are what allow a program to respond intelligently to different situations rather than doing the same thing every time.

4.1 Learning Outcomes

- LO 4.1 Plan decision-making logic for a problem.
- LO 4.2 Make decisions with if and if-else structures.
- LO 4.3 Use chained if-else-if, the conditional operator, and recognize the switch structure.

4.2 Introduction

Real programs must respond differently to different situations. A grading program assigns different letters for different scores; a login system admits some users and rejects others; a game reacts differently to each move. This branching behavior is achieved with decision control structures, which evaluate a condition (a Boolean expression that is either true or false) and choose which statements to execute based on the result. This chapter introduces Java's decision structures and the disciplined thinking required to use them well.

Decision-making rests directly on the relational and logical operators from Chapter 2, since a decision's condition is a Boolean expression built from them. Planning the logic before writing the code, thinking clearly about exactly which cases lead to which actions, is as important as the syntax, because a decision structure that is syntactically correct but logically wrong produces a program that runs but does the wrong thing, often in ways that are hard to spot.

4.3 Historical Background

The if-then-else construct is one of the oldest and most universal ideas in programming, present in essentially every language since the earliest days of high-level programming in the 1950s. Its universality reflects a fundamental truth: any non-trivial computation must at some point choose between alternatives based on data. The structured programming movement of the 1960s and 1970s established that clear, well-nested decision structures, rather than unrestricted jumps between parts of a program, were essential to writing code that humans could understand and maintain.

Java inherited this structured tradition directly, providing clean if-else and switch constructs while omitting the unrestricted goto jump that structured programming had discredited. This is part of a consistent theme in Java's design that this book returns to repeatedly: the language deliberately steers programmers toward clear, disciplined structure, trusting that readable, maintainable code matters more over the life of real software than the freedom to write cleverly compressed but obscure logic.

4.4 Definitions

KEY TERMS

Decision Control Structure: A construct that chooses which statements to execute based on a condition.

Condition: A boolean expression, evaluating to true or false, that a decision structure tests.

if-else Structure: A structure that executes one block when a condition is true and another when it is false.

Conditional (Ternary) Operator: The ?: operator is a compact way to choose between two values based on a condition.

> **switch Structure:** A structure that selects among several branches based on the value of a single expression.

4.5 Core Concepts of Decision Making

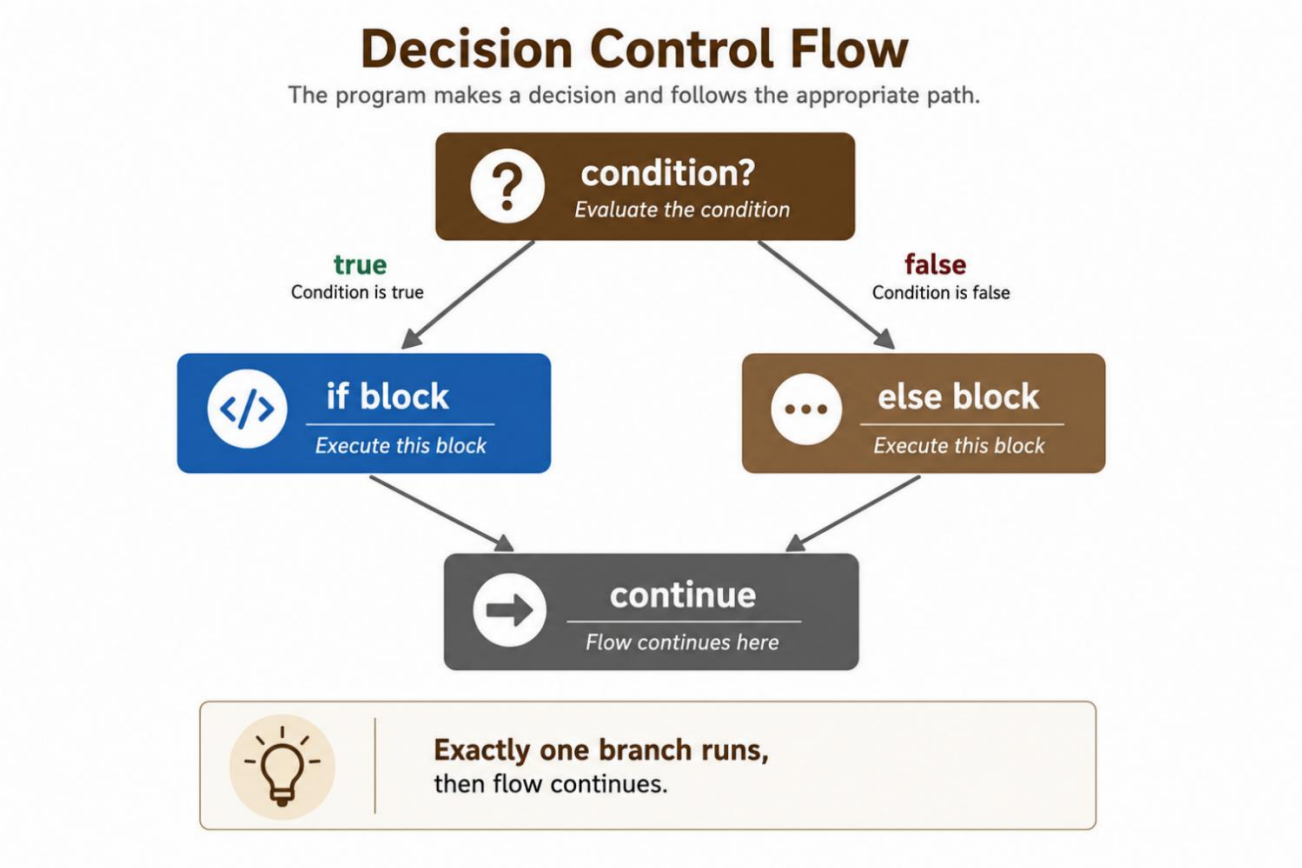

Figure 4.1. An if-else decision runs exactly one branch based on a condition, then flow continues.

4.5.1 if and if-else

The simplest decision is the if statement: it executes a block of statements only when its condition is true, and otherwise skips it. Adding an else clause provides an alternative block to execute when the condition is false, so that exactly one of the two blocks always runs. The condition is a boolean expression, typically built with the relational operators (such as >=) and logical operators (such as &&) from Chapter 2. The braces that enclose each block group its statements together, and using them consistently, even for single statements, prevents a common class of errors.

4.5.2 Chained Decisions and the Conditional Operator

When there are more than two possibilities, if-else structures can be chained: an if, followed by one or more else if clauses, followed optionally by a final else. Java evaluates each condition in order and executes the block for the first one that is true, skipping the rest. This chained form is ideal for classifying a value into ranges, such as converting a numeric score into a letter grade. For the narrower case of choosing between just two values, the conditional operator (condition ? valueIfTrue : valueIfFalse) offers a compact expression that yields one value or the other.

4.5.3 A Worked Example

The program below classifies a numeric score into a letter grade using a chained if-else-if structure, then uses the conditional operator to derive a pass-or-fail status. It demonstrates both decision forms working together on a realistic task.

```
public class Decisions {
    public static void main(String[] args) {
        int score = 82;
        if (score >= 90) {
            System.out.println("Grade: A");
        } else if (score >= 80) {
            System.out.println("Grade: B");
        } else if (score >= 70) {
            System.out.println("Grade: C");
        } else {
            System.out.println("Grade: F");
        }
        String status = (score >= 75) ? "Passed" : "Failed";
        System.out.println("Status: " + status);
    }
}
```

Listing 4.1. Decisions with if-else-if and the conditional operator. Verified output: Grade: B / Status: Passed

Trace the logic: with a score of 82, the first condition (score >= 90) is false, so Java moves to the next; the second (score >= 80) is true, so it prints Grade: B and skips the remaining clauses entirely. The order of the conditions matters here: because each range builds on the previous one being false, listing them from highest to lowest is what makes the chain correct. Getting this ordering wrong is a classic logic error that produces a program which compiles and runs but assigns the wrong grades.

4.6 Table 4.1, Decision Structures

Structure	Best For	Result
if	A single optional action	Runs a block only when true
if-else	Two alternatives	Runs exactly one of two blocks
if-else-if chain	Several ranges or cases	Runs the first matching block
Conditional operator	Choosing between two values	Yields one value or the other
switch	Many discrete values of one expression	Branches on the matched value

Table 4.1. Java's decision structures at a glance.

4.7 The switch Structure and Planning Logic

When a decision selects among many discrete, specific values of a single expression, such as a menu choice of 1, 2, 3, or 4, the switch structure can express the logic more clearly than a long if-else chain. It compares one expression against a series of case values and executes the matching branch. Beyond syntax, the deeper skill this chapter teaches is planning the decision logic before coding: enumerating exactly which conditions lead to which actions, checking that every case is covered and none overlap incorrectly, and confirming that the ordering of chained conditions is right. A few minutes of clear planning prevents logic errors that can be surprisingly hard to find once buried in code.

4.8 Philippine Context and Case Study

PHILIPPINE CONTEXT

Decision logic is at the heart of the business rules that Philippine enterprise and financial software must implement precisely, from tax bracket computations and loan eligibility rules to tiered pricing and grading systems in educational software. Errors in this logic, a misplaced boundary between brackets, a wrongly ordered chain of conditions, translate directly into incorrect real-world outcomes, which is why careful, well-planned decision structures are a practical professional necessity in the Philippine software industry, not merely an academic exercise.

CASE STUDY

Illustrative composite case.

A Philippine payroll program computed tax using a chained if-else structure, but a developer had ordered the brackets from lowest to highest rather than highest to lowest, so nearly every employee matched the first, lowest bracket and was under-taxed. The code compiled and ran without error; the bug was purely a matter of logic and ordering. Reordering the conditions from the highest bracket to the lowest fixed it immediately. The incident illustrates this chapter's central caution: a decision structure can be perfectly valid Java and still be logically wrong, so planning and tracing the logic matter as much as the syntax.

4.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Plan the decision logic, enumerating cases and their actions, before writing the code.
- Best Practice: Always use braces around decision blocks, even single statements, to prevent structural errors.
- Best Practice: Order chained conditions carefully, since Java executes the first matching block and skips the rest.
- Common Pitfall: Ordering a chained if-else-if incorrectly, so an earlier condition wrongly captures cases meant for a later one.
- Common Pitfall: Using = (assignment) where == (comparison) was intended in a condition.
- Common Pitfall: Omitting braces and accidentally associating only the first statement with the decision.

4.10 Summary and Key Takeaways

Decision control structures let a program choose different actions based on a condition, a Boolean expression that is true or false. The if and if-else structures handle one or two alternatives; chained if-else-if handles several ranges or cases; the conditional operator compactly chooses between two values; and switch selects among many discrete values. Planning the logic and ordering chained conditions correctly matters as much as the syntax. Chapter 5 now turns to repeating actions with loops.

KEY TAKEAWAYS

- Decision structures choose actions based on a condition (a Boolean expression).
- if-else handles two alternatives; chained if-else-if handles several; switch handles many discrete values.
- Java runs the first matching block in a chain and skips the rest, so ordering matters.
- A decision structure can be valid Java yet logically wrong; plan and trace the logic.

4.11 Key Terms, Reflection, and Discussion

Decision Control Structure. Condition. if. if-else. else if. Conditional Operator. switch. Boolean Expression.

- Reflection: What decisions will your Application Build need to make, and what conditions govern each?
- Discussion: The conditional operator can compress a small if-else into one line. When does this brevity aid readability, and when does it harm it?

4.12 Activity and Application Build, Step 4

APPLICATION BUILD, STEP 4

Identify the decisions your application must make (for example, classifying a grade, checking whether an account has sufficient balance, or validating a menu choice). Write and test at least one decision structure implementing a real rule of your application, planning the logic in words first, then translating it into an if-else or switch structure.

4.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. The condition tested by an if statement must evaluate to:

a) An integer b) A boolean (true or false) c) A string d) A double

2. In a chained if-else-if, Java executes:

a) Every matching block b) The first matching block only c) The last matching block d) No blocks

3. The conditional operator ?: is best used to:

a) Loop over values b) Choose between two values c) Read input d) Declare a class

4. The switch structure is best suited for:

a) Two alternatives b) Many discrete values of one expression c) Repeating an action d) Reading input

5. Ordering conditions from the highest range to the lowest in a grade classifier matters because:

a) Java is faster that way b) Java runs the first true block and skips the rest c) It uses less memory d) switch requires it

4.14 References

- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
- Oracle. The Java tutorials: Control flow statements. Oracle Corporation.
- Schildt, H. (2021). Java: The complete reference (12th ed.). McGraw-Hill.

End of Chapter 4. Continue to Chapter 5: Loops and Repetition

LOOPS AND REPETITION

Doing Something Many Times Without Repeating Yourself

"The loop is where a program stops being a list of instructions and starts being able to do arbitrarily much work with arbitrarily little code."

5.0 Chapter Overview

This chapter introduces loops, the control structures that repeat a block of statements. You will learn Java's three loop structures, while, do-while, and for, when each is most appropriate, and how nested loops handle two-dimensional problems. Loops are what let a short program process a large amount of data, and together with the decisions of Chapter 4 they complete the control-flow toolkit that every program relies on.

5.1 Learning Outcomes

- LO 5.1 Explain the structure and purpose of a loop.
- LO 5.2 Write programs using while, do-while, and for loop structures.
- LO 5.3 Use nested loops and choose the appropriate loop for a task.

5.2 Introduction

Much of what programs do involves repetition: summing a list of numbers, printing each item in a collection, retrying until valid input is given, processing every record in a file. Writing the same statement over and over would be impractical and would not work when the number of repetitions is not known in advance. Loops solve this by executing a block of statements repeatedly, as long as a condition holds. A single well-written loop can process five items or five million with exactly the same code, which is what gives programs their power to handle real quantities of data.

Java provides three loop structures, each suited to a slightly different situation, and this chapter examines all three along with the nested loops used for grid-like problems. Loops build

directly on the conditions from Chapter 4, since every loop is governed by a condition that determines whether to continue, and getting that condition right, so the loop neither stops too early nor runs forever, is the central skill this chapter develops.

5.3 Historical Background

Iteration has been fundamental to computing since its origins; the very idea of a stored program that could repeat instructions was central to the earliest computer designs. As structured programming matured, the while and for loops emerged as the disciplined, readable ways to express repetition, replacing the error-prone manual jumps that early programmers had used. These structured loops made it possible to reason clearly about repetition, in particular about the conditions under which a loop starts, continues, and stops.

Java inherited the C-family loop syntax, which by the time of Java's creation was already familiar to a generation of programmers, ensuring that the while, do-while, and for structures felt natural to those coming from other languages. The consistency of this loop syntax across many languages is one reason that, once you understand loops in Java, the same understanding transfers readily to most other modern programming languages you may encounter.

5.4 Definitions

KEY TERMS

Loop: A control structure that repeats a block of statements while a condition holds.

while Loop: A loop that tests its condition before each iteration, and may run zero times.

do-while Loop: A loop that tests its condition after each iteration, and always runs at least once.

for Loop: A loop that gathers initialization, condition, and update into one compact header, ideal for counting.

Nested Loop: A loop placed inside another loop, used for two-dimensional or grid-like repetition.

5.5 Core Concepts of Loops

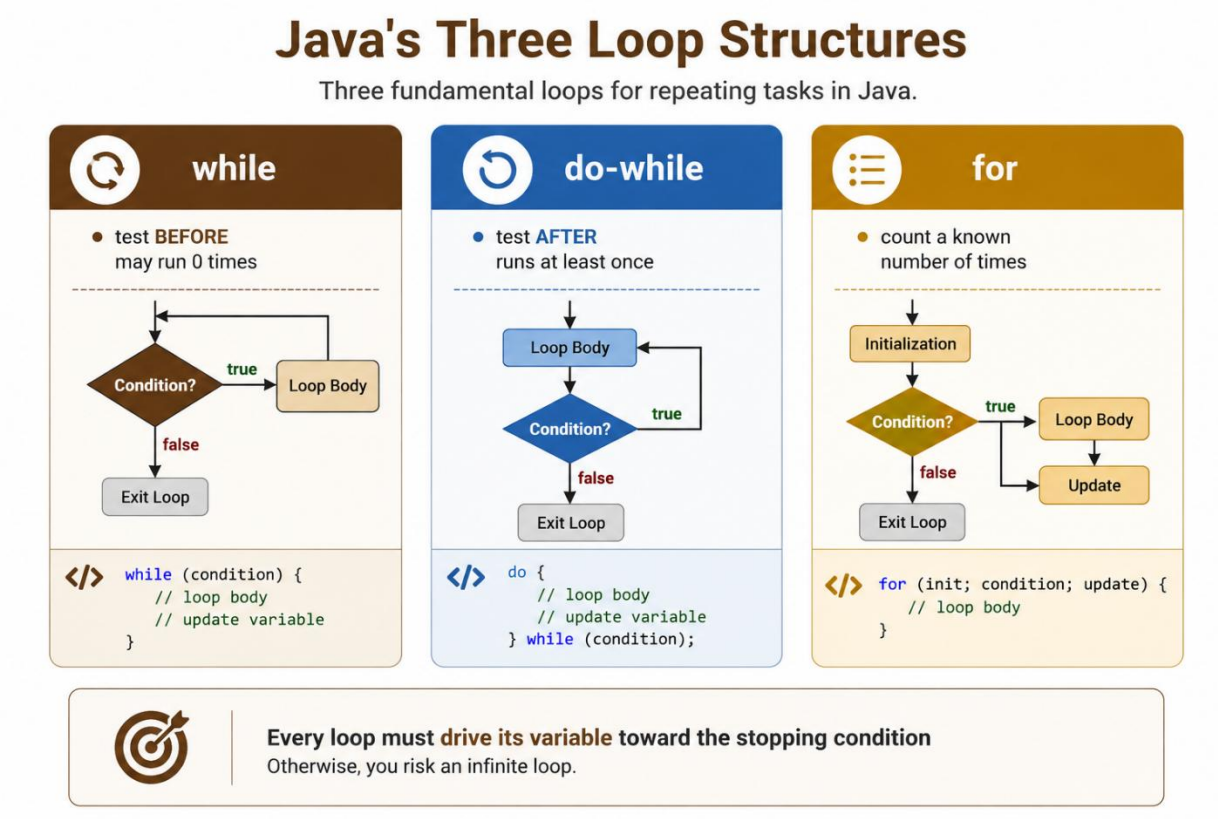

Figure 5.1. Java's three loop structures and when each is most appropriate.

5.5.1 The Three Loop Structures

The while loop tests its condition before each iteration and repeats as long as the condition is true; if the condition is false at the start, the loop body runs zero times. The do-while loop tests its condition after each iteration, so its body always runs at least once, which suits situations such as prompting for input that must happen at least once. The for loop gathers three parts into a single compact header: an initialization (run once at the start), a condition (tested before each iteration), and an update (run after each iteration). The for loop is ideal when the number of repetitions is known or countable.

5.5.2 A Worked Example

The program below uses a for loop to sum the integers from one to five, then a while loop to count down from three. It demonstrates the two most common loop forms side by side.

```
public class Loops {
    public static void main(String[] args) {
        int total = 0;
        for (int i = 1; i <= 5; i++) {
```

```
            total = total + i;
        }
        System.out.println("Sum 1 to 5 is " + total);
        int n = 3;
        while (n > 0) {
            System.out.println("Countdown: " + n);
            n--;
        }
    }
}
```

Listing 5.1. A for loop and a while loop. Verified output: Sum 1 to 5 is 15 / Countdown: 3 / Countdown: 2 / Countdown: 1

In the for loop, i starts at 1, the loop continues while i is at most 5, and i increases by one after each pass, so the body runs for i equal to 1, 2, 3, 4, and 5, accumulating their sum. In the while loop, n starts at 3 and the body decreases it by one each time (n--), so the loop prints 3, 2, and 1 and then stops when n reaches 0. The essential discipline in both is ensuring the loop variable changes toward the stopping condition; a loop whose variable never reaches the condition runs forever.

5.5.3 Nested Loops

A loop may contain another loop, forming a nested loop, which is the natural way to process two-dimensional structures such as grids, tables, or the rows and columns of a multiplication table. The outer loop advances once for every complete run of the inner loop, so a pair of nested loops each running five times executes the innermost body twenty-five times. Nested loops are powerful but must be used thoughtfully, since the total work multiplies, and deeply nested loops over large data can become slow.

5.6 Table 5.1, The Three Loops Compared

Loop	Condition Tested	Best For
while	Before each iteration	Repeating an unknown number of times
do-while	After each iteration	When the body must run at least once
for	Before each iteration	Counting a known number of iterations

Loop	Condition Tested	Best For
Nested	Each loop's own condition	Two-dimensional or grid problems

Table 5.1. Choosing among Java's loop structures.

5.7 Avoiding Infinite Loops

The single most important discipline with loops is ensuring they actually terminate. A loop continues as long as its condition is true, so if nothing in the loop body ever makes the condition false, the loop runs forever, an infinite loop that hangs the program. Every loop should be written so that its controlling variable moves steadily toward the stopping condition: a counter that increments toward its limit, a value that decrements toward zero, or a flag that will eventually be set. When a loop misbehaves, the first thing to check is whether its condition can ever become false, and whether the loop body actually changes the variable the condition depends on.

5.8 Philippine Context and Case Study

PHILIPPINE CONTEXT

Loops underlie the batch processing that Philippine enterprise systems perform constantly, iterating over every employee to compute payroll, every transaction to generate a report, or every student record to produce grades. In these settings, a loop that terminates incorrectly, or one nested inefficiently over large data, has direct operational consequences, from a hung overnight batch job to a report that takes hours instead of minutes. Writing correct, efficient loops is therefore a practical, everyday competency in the Philippine software industry.

CASE STUDY

Illustrative composite case.

A student wrote a while loop to keep prompting for input until the user entered a valid value, but forgot to update the variable the condition depended on inside the loop,

> so the condition never changed and the program repeated the same prompt endlessly. The fix was to ensure the loop body read a new value each pass, moving the loop toward its stopping condition. The experience taught the lesson this chapter stresses: every loop must contain something that drives its controlling variable toward the condition that ends it, or it will never stop.

5.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Ensure every loop's controlling variable moves toward its stopping condition, so the loop terminates.
- Best Practice: Use a for loop when the number of iterations is known, and a while loop when it is not.
- Best Practice: Be cautious with nested loops over large data, since the total work multiplies.
- Common Pitfall: Writing an infinite loop by never changing the variable its condition depends on.
- Common Pitfall: Off-by-one errors, running a loop one time too many or too few by mis-setting the condition.
- Common Pitfall: Choosing do-while when the body should sometimes run zero times, or while when it must always run once.

5.10 Summary and Key Takeaways

Loops repeat a block of statements while a condition holds. The while loop tests before iterating and may run zero times; the do-while loop tests after and always runs at least once;

the for loop compactly counts a known number of iterations. Nested loops handle two-dimensional problems. The essential discipline is ensuring every loop terminates by driving its controlling variable toward the stopping condition. Chapter 6 now turns to arrays, which store many values that loops naturally process.

KEY TAKEAWAYS

- Loops repeat statements while a condition holds; while, do-while, and for are the three structures.
- while tests before (may run zero times); do-while tests after (runs at least once); for counts.
- Nested loops handle two-dimensional problems, but the total work multiplies.
- Every loop must drive its controlling variable toward the stopping condition, or it runs forever.

5.11 Key Terms, Reflection, and Discussion

Loop. while Loop. do-while Loop. for Loop. Nested Loop. Infinite Loop. Iteration. Loop Condition. Off-by-One Error.

- Reflection: Where will your Application Build need to repeat an action, over a list of records, until valid input, across a menu? Which loop fits each case?
- Discussion: The for and while loops can each be rewritten as the other. Given that, why does Java (and most languages) provide both?

5.12 Activity and Application Build, Step 5

APPLICATION BUILD, STEP 5

Add a loop to your application. For example, write a menu that repeats until the user chooses to exit, or a loop that processes several records in sequence. Choose the loop type deliberately, justify the choice, and confirm that the loop terminates correctly under all inputs you can think of.

5.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. Which loop always executes its body at least once?

a) while b) do-while c) for d) None

2. The for loop's header contains, in order:

a) Condition, update, initialization b) Initialization, condition, update c) Update, condition, initialization d) Only a condition

3. A loop that never terminates is called:

a) A nested loop b) An infinite loop c) A for loop d) A do-while loop

4. Nested loops are most useful for:

a) Reading input b) Two-dimensional or grid problems c) Making decisions d) Declaring variables

5. The first thing to check when a loop runs forever is:

a) The variable types b) Whether the condition can ever become false c) The class name d) The import statements

5.14 References

- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
- Oracle. The Java tutorials: The for, while, and do-while statements. Oracle Corporation.
- Schildt, H. (2021). Java: The complete reference (12th ed.). McGraw-Hill.

End of Chapter 5. Continue to Chapter 6: Arrays

ARRAYS

Storing and Processing Many Values Together

"One variable holds one thing. An array holds a hundred, or a million, and lets a single loop touch each of them."

6.0 Chapter Overview

This chapter closes Part II by introducing arrays, which store a fixed number of values of the same type under a single name. You will learn to declare and create arrays, access their elements by index, determine their length, and use multidimensional arrays for grid-like data. Arrays are the first data structure this book covers, and combined with the loops of Chapter 5, they let a program process a collection of data with concise, general code, a capability that objects will build upon in the chapters ahead.

6.1 Learning Outcomes

- LO 6.1 Declare and create arrays.
- LO 6.2 Access array elements by index and determine an array's length.
- LO 6.3 Declare and use multidimensional arrays.

6.2 Introduction

So far, each variable has held a single value. But programs constantly deal with collections: the scores of a class, the items in an order, the days of a month. Declaring a separate variable for each would be impractical and would not work when the number of values is not known in advance. An array solves this by holding many values of the same type under one name, with each value, called an element, accessible by its position, called its index. Arrays and loops are natural partners: a loop can step through an array's indices to process every element with just a few lines of code.

This chapter introduces arrays as the first step toward the richer data structures and objects that follow. An array is a fixed-size, ordered collection of the same-typed values, and understanding how to create one, access its elements safely, and iterate over it is a prerequisite for nearly everything ahead, since objects in later chapters will frequently hold arrays, and collections of objects will frequently be stored in them.

6.3 Historical Background

The array is one of the oldest and most fundamental data structures in computing, present since the earliest programming languages because it maps so directly onto how computer memory is organized: a contiguous block of storage, each slot holding one value, addressable by its position. This direct correspondence to memory is why array access by index is extremely fast, and why the array remains a foundational building block beneath even the most sophisticated modern data structures.

Java refined the array with an important safety feature: it tracks each array's length and checks every access against it, so that an attempt to read or write beyond the array's bounds is caught and reported rather than silently corrupting other memory, as could happen in some older languages. This bounds checking, part of Java's broader emphasis on catching errors safely, prevents an entire category of dangerous bugs, at the modest cost of the check itself, a trade-off consistent with Java's design priorities throughout.

6.4 Definitions

KEY TERMS

Array: A fixed-size, ordered collection of values of the same type, stored under a single name.

Element: One value stored in an array.

Index: The position of an element in an array, starting at 0 for the first element.

Length: The number of elements an array holds, available as the array's length field.

Multidimensional Array: An array of arrays, used to represent grids or tables of values.

6.5 Core Concepts of Arrays

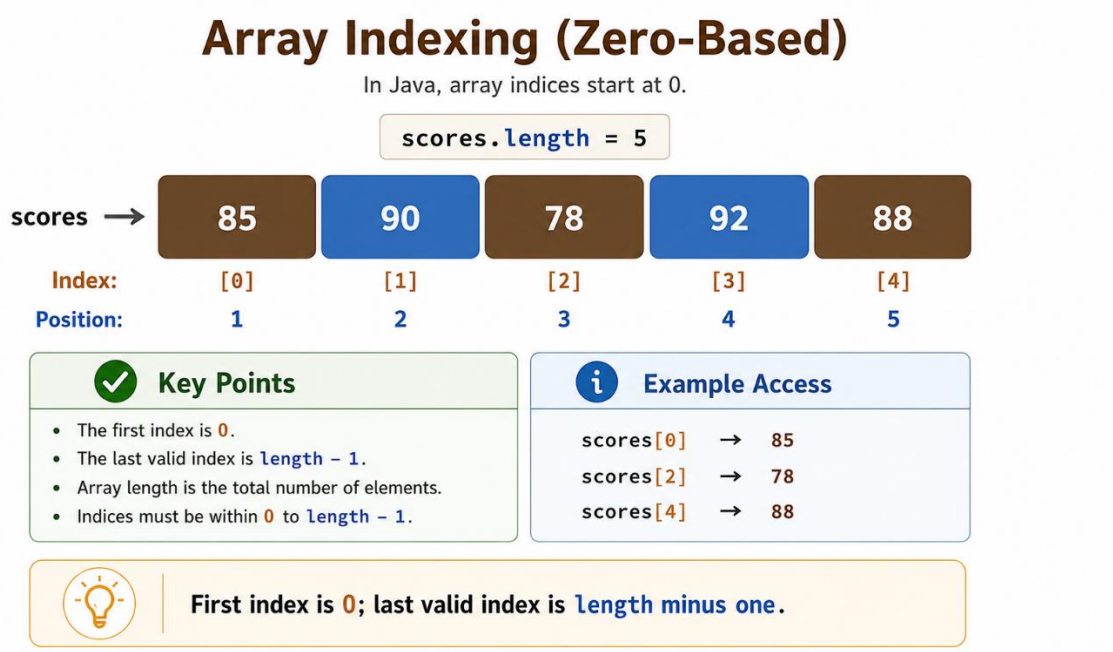

Figure 6.1. Zero-based array indexing: the first element is at index 0 and the last at length minus one.

6.5.1 Declaring, Creating, and Accessing

An array is declared with a type followed by square brackets, and can be created and filled at once with an initializer list in braces, as in int[] scores = {85, 90, 78};. Each element is accessed by its index in square brackets, and a crucial rule governs indexing: the first element is at index 0, not 1, so an array of five elements has valid indices 0 through 4. This zero-based indexing is universal in Java and most modern languages, and remembering it prevents the most common array error, reaching beyond the last valid index.

6.5.2 Length and Iteration

Every array knows its own size through its length field, accessed as arrayName.length. This is what makes arrays and loops such natural partners: a for loop can run its index from 0 up to, but not including, length, visiting every element exactly once regardless of the array's size. Using length rather than a hard-coded number means the same loop works correctly if the array's size changes, an important habit for writing general, reusable code. The example below sums an array of scores and computes their average using exactly this pattern.

```
public class Arrays1 {
    public static void main(String[] args) {
        int[] scores = {85, 90, 78, 92, 88};
        int total = 0;
```

```
        for (int i = 0; i < scores.length; i++) {
            total = total + scores[i];
        }
        double average = (double) total / scores.length;
        System.out.println("Number of scores: " + scores.length);
        System.out.println("Average score: " + average);
    }
}
```

Listing 6.1. Summing and averaging an array. Verified output: Number of scores: 5 / Average score: 86.6

Study the loop condition i < scores.length: it uses less-than, not less-than-or-equal, precisely because the valid indices run from 0 to length minus one. The expression (double) total / scores.length uses a cast, (double), to convert the integer total to a floating-point value before dividing, ensuring a fractional average rather than the truncated integer division of Chapter 2. Both details, the zero-based bound and the cast for a fractional result, recur constantly in array processing.

6.5.3 Multidimensional Arrays

A multidimensional array, most commonly two-dimensional, is an array of arrays, used to represent grids, tables, or matrices. A two-dimensional array is accessed with two indices, one for the row and one for the column, and is naturally processed with nested loops, the outer loop over rows and the inner over columns, connecting directly to the nested loops of Chapter 5. Two-dimensional arrays model anything with a natural grid structure, from a game board to a spreadsheet of values.

6.6 Table 6.1, Array Essentials

Concept	Detail	Note
First index	0	Zero-based; last index is length minus one
Size	arrayName.length	A field, not a method
Element type	All the same	An array holds one type only
Size after creation	Fixed	Cannot grow; create a new array to resize
Bounds	Checked	Out-of-range access is caught and reported

Table 6.1. Key facts about Java arrays.

6.7 Arrays as a Bridge to Objects

Arrays matter to this book beyond their own usefulness because they are a bridge to the object-oriented material ahead. Once you can create objects in Chapter 7 and beyond, you will frequently store many of them in an array: an array of Student objects, an array of Account objects. The same declaring, indexing, and iterating you learn here applies directly, with an object type in place of a primitive. Arrays are also the simplest example of a broader idea, the data structure, that organizes multiple values so a program can process them systematically, an idea that grows richer throughout your study of computing. For now, mastering the array's basics equips you to handle collections in every chapter that follows.

6.8 Philippine Context and Case Study

PHILIPPINE CONTEXT

Arrays and the iteration over them underlie the data processing at the core of Philippine business software, from computing statistics over a set of sales figures to tabulating survey responses or managing rows of records. A developer's fluency with safe, correct array processing, respecting bounds, using length, iterating cleanly, directly affects the correctness of these everyday computations, making it a foundational practical skill across the country's software sector.

CASE STUDY

Illustrative composite case.

A student processing an array of test scores wrote a loop whose condition used less-than-or-equal against the array's length, causing the loop to reach one position past the last valid element and triggering an out-of-bounds error the moment it ran. Because Java checks array bounds, the error was reported clearly rather than silently corrupting data, and changing the condition to less-than fixed it at once. The incident illustrates both the zero-based indexing rule this chapter stresses and the value of Java's bounds checking, which turned a potentially dangerous mistake into an immediately visible, easily corrected one.

6.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Iterate with a for loop from 0 up to (but not including) arrayName.length, so the code adapts to any size.
- Best Practice: Remember that arrays are zero-based; the last valid index is length minus one.
- Best Practice: Cast to double when averaging integer arrays, to avoid integer-division truncation.
- Common Pitfall: Using <= length in a loop condition, reaching one past the last valid index, and causing an out-of-bounds error.
- Common Pitfall: Expecting an array to grow after creation; its size is fixed once created.
- Common Pitfall: Forgetting that length is a field (arrayName.length), not a method call.

6.10 Summary and Key Takeaways

An array stores a fixed number of same-typed values under one name, each accessed by a zero-based index, with the count available through the length field. Arrays and loops are natural partners, allowing a single loop to process collections of any size. Multidimensional arrays represent grids and are processed with nested loops. Java checks array bounds, catching out-of-range access safely. Arrays bridge to the objects ahead, which are often stored in arrays. This chapter closes Part II; Part III now turns to objects and classes, the heart of object-oriented programming.

KEY TAKEAWAYS

- An array holds a fixed number of same-typed values, accessed by zero-based index.

- arrayName.length gives the size; loop from 0 up to (not including) length.
- Multidimensional arrays represent grids and are processed with nested loops.
- Java checks array bounds, catching out-of-range access rather than corrupting memory.

6.11 Key Terms, Reflection, and Discussion

Array. Element. Index. Length. Zero-Based Indexing. Multidimensional Array. Bounds Checking. Cast.

- Reflection: Where will your Application Build hold a collection of values or, later, objects? Would an array serve, and what would its element type be?
- Discussion: Java arrays have a fixed size once created. What are the advantages of this simplicity, and what problems does it create that richer collections later solve?

6.12 Activity and Application Build, Step 6

APPLICATION BUILD, STEP 6

Add an array to your application to hold a collection of related values, such as scores, prices, or names. Write a loop that processes the array, computing a total, an average, a maximum, or a count, and display the result. Use arrayName.length in your loop so the code adapts if the collection's size changes.

6.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. The first element of a Java array is at index:

a) 1 b) 0 c) -1 d) length

2. The number of elements in an array named data is given by:

a) data.size() b) data.length c) data.count d) length(data)

3. For an array of 5 elements, the last valid index is:

a) 5 b) 4 c) 6 d) 0

4. A two-dimensional array is naturally processed with:

a) A single while loop b) Nested loops c) No loops d) A switch

5. Accessing an array beyond its bounds in Java:

a) Silently corrupts memory b) Is caught and reported as an error c) Returns zero d) Grows the array

6.14 References

- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
- Oracle. The Java tutorials: Arrays. Oracle Corporation.
- Schildt, H. (2021). Java: The complete reference (12th ed.). McGraw-Hill.

End of Chapter 6. Continue to Chapter 7: Objects and Classes

PART III

CLASSES AND OBJECTS

Part III reaches the heart of object-oriented programming. It builds from the fundamental class-object distinction through methods and the discipline of encapsulation to complete, well-formed classes with constructors, overloading, and packages, the foundation on which the object-oriented pillars are built.

OBJECTS AND CLASSES

The Heart of Object-Oriented Programming

"A class is a noun the program can reason about: not just data, and not just behavior, but a thing that has both."

7.0 Chapter Overview

Parts I and II built the foundations: data, control flow, and arrays. Part III now reaches the heart of object-oriented programming, the class and the object. This chapter explains the difference between a class and an object, how to define a class with fields and methods, how to create objects from it, and the crucial distinction between instance members, which belong to each object, and static members, which belong to the class as a whole. This is the conceptual center of the entire course, and the material here underlies everything that follows.

7.1 Learning Outcomes

- LO 7.1 Explain object-oriented programming and differentiate between classes and objects.
- LO 7.2 Differentiate instance variables and methods from static (class) variables and methods.
- LO 7.3 Create objects from a class and call their methods.

7.2 Introduction

Chapter 1 introduced the idea that object-oriented programming organizes a program around objects that bundle data with behavior. This chapter makes that idea concrete. A class defines a new type: it specifies what data every object of that type will hold (its fields, also called instance variables) and what it can do (its methods). An object is a specific instance of a class, created at runtime, with its own copy of the data. If the class Student is the blueprint,

then each individual student, Ana, Ben, Carla, is an object built from it, each holding her or his own name, identification number, and grade.

Understanding the class-object relationship clearly is the single most important step in learning object-oriented programming, because every later concept, encapsulation, inheritance, and polymorphism, builds directly upon it. This chapter, therefore, takes care to distinguish the class (one blueprint) from the objects (many instances) and to introduce the important distinction between members that belong to each object and members that belong to the class as a whole.

7.3 Historical Background

The concepts of class and object originated in the 1960s with the Simula language, designed for simulation, where it was natural to model real-world entities as objects with their own state and behavior. In the 1970s, Smalltalk further developed these ideas, establishing object-oriented programming as a distinct paradigm. When Java arrived in the 1990s, it adopted the class as the mandatory organizing unit of every program, making the class-object model not merely available but unavoidable, and thereby teaching a generation of programmers to think in terms of objects by default.

This history matters because it explains why object-oriented programming feels so natural for modeling real problems: it was invented precisely to model real-world entities in simulations. When you define a Student, Account, or Book class, you are doing exactly what the paradigm's inventors intended: capturing a real-world thing as a program construct that holds its own data and knows its own behavior. Keeping this modeling purpose in mind helps you design classes that correspond cleanly to the concepts in your problem.

7.4 Definitions

KEY TERMS

Class: A blueprint that defines the data (fields) and behavior (methods) of a type of object.

Object: A specific instance of a class, created at runtime, holding its own copy of the instance data.

Instance Variable (Field): A variable that belongs to each object individually, holding that object's own state.

Instance Method: A method that operates on a particular object's data, called through that object.

Static (Class) Member: A variable or method that belongs to the class as a whole, shared across all objects rather than held per object.

7.5 Core Concepts of Classes and Objects

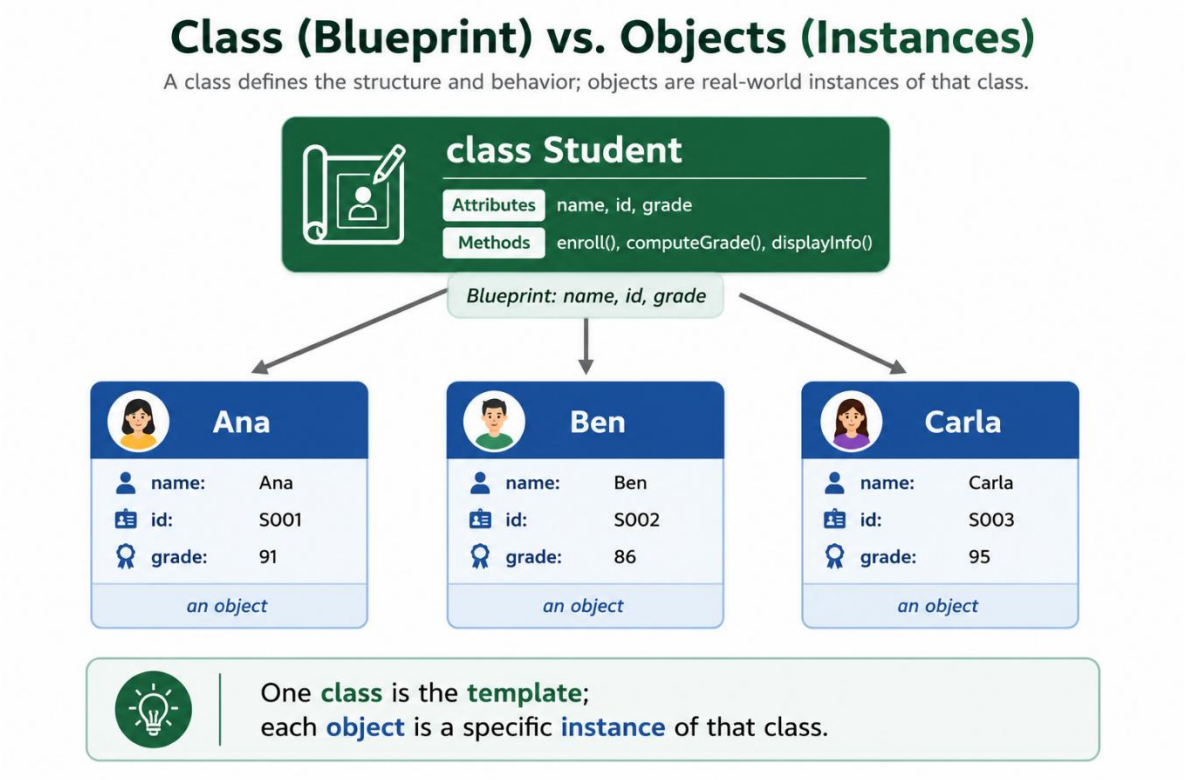

Figure 7.1. One class is a blueprint; each object is a specific instance created from it.

7.5.1 Defining a Class and Creating Objects

A class is defined with the keyword class followed by its name and a body containing its fields and methods. To create an object from a class, you use the new keyword, which allocates a new instance and returns a reference to it that you store in a variable. Once you have an object, you access its fields and call its methods using the dot operator. The program below defines a simple Student class with three fields and a display method, then creates a Student object, sets its fields, and calls its method.

```
public class StudentDemo {
    static class Student {
        String name;
        int id;
        double grade;
        void display() {
            System.out.println("Student(id=" + id + ", name=" + name + ", grade=" + grad
e + ")");
        }
    }
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.name = "Ana";
        s1.id = 101;
        s1.grade = 88.5;
        s1.display();
    }
}
```

Listing 7.1. Defining a class and creating an object. Verified output: Student(id=101, name=Ana, grade=88.5)

The Student class here defines the blueprint: every Student object will have a name, an id, a grade, and the ability to display itself. The statement new Student() creates one actual object, and s1 holds a reference to it. Setting s1.name, s1.id, and s1.grade fills in that particular object's data, and s1.display() calls the method on that object, which prints that object's own values. If a second Student object were created, it would have its own independent copies of these fields. (The class is written as a nested static class here purely so the example runs as a single file; in practice each class usually lives in its own file, as Chapter 9 discusses.)

7.5.2 Instance vs. Static Members

A crucial distinction governs what a field or method belongs to. An instance member, the default, belongs to each object individually: every Student object has its own name and grade, and calling display on one object shows that object's data. A static member, marked with the static keyword, belongs to the class as a whole and is shared across all objects: a static counter tracking how many students have been created, for example, is one value shared by the entire class, not a separate value per object. The rule of thumb is that data unique to each object is an instance variable, while data or behavior belonging to the type as a whole is static.

7.5.3 Objects Are References

An important subtlety is that a variable of a class type does not hold the object itself, but a reference to it, a way of locating the object in memory. This has practical consequences: assigning one object variable to another makes both refer to the same object, not two copies,

so a change made through one is visible through the other. Similarly, comparing two object references with == checks whether they refer to the very same object, not whether two distinct objects have equal contents. This reference nature of objects is a frequent source of early confusion, and being aware of it now prevents a class of puzzling bugs later.

7.6 Table 7.1, Instance vs. Static

Aspect	Instance Member	Static Member
Belongs to	Each object individually	The class as a whole
Copies	One per object	One shared by all objects
Accessed through	An object reference	The class name
Example	A student's own grade	A count of all students

Table 7.1. Instance members versus static members.

7.7 Designing Good Classes

A well-designed class models a single, coherent concept, with fields that represent that concept's data and methods that represent its behavior. A Student class should hold student data and student behavior, not unrelated concerns; mixing unrelated responsibilities into one class is a common design flaw that makes code harder to understand and change. As you design the classes for your Application Build, ask of each field and method whether it genuinely belongs to the concept the class represents. This discipline, one class for one clear concept, is the foundation of good object-oriented design, and it pays off increasingly as programs grow and the inheritance and polymorphism of Part IV come into play.

7.8 Philippine Context and Case Study

> **PHILIPPINE CONTEXT**
>
> Object-oriented design skill, the ability to model a real domain as a well-structured set of classes, is exactly what Philippine software employers seek beyond mere syntax knowledge, since it determines whether a developer can contribute to the large, long-

lived enterprise systems common in the country's IT and outsourcing sectors. A graduate who can look at a business domain, students, accounts, products, transactions, and design clean classes that model it faithfully holds a competency that distinguishes a software engineer from someone who merely writes code.

CASE STUDY

Illustrative composite case.

A Philippine development team modeling a school system initially crammed student data, enrollment logic, and report formatting all into one large class, which quickly became difficult to understand and modify. Redesigning around distinct classes, a Student holding student data and behavior, separate classes for enrollment and reporting, gave each class one clear responsibility and made the system far easier to work with. The team's experience illustrates this chapter's central design lesson: a class should model one coherent concept, and honoring that principle from the start prevents the tangled, hard-to-change code that ignoring it produces.

7.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Design each class to model a single, coherent concept, with fields and methods that genuinely belong to it.
- Best Practice: Use instance members for data unique to each object, and static members for what belongs to the class as a whole.
- Best Practice: Remember that object variables hold references, so assignment and == compare references, not contents.
- Common Pitfall: Confusing a class (the blueprint) with an object (a specific instance created from it).

- Common Pitfall: Making a member static when each object should have its own copy, or the reverse.
- Common Pitfall: Comparing objects with == expecting content equality, when it actually compares references.

7.10 Summary and Key Takeaways

A class is a blueprint defining the fields and methods of a type of object, and an object is a specific instance created from it with the new keyword, holding its own copy of the instance data. Instance members belong to each object individually, while static members belong to the class as a whole and are shared. Object variables hold references, not objects themselves, which affects how assignment and comparison behave. Good design gives each class one coherent concept. Chapter 8 now examines methods and encapsulation in depth.

KEY TAKEAWAYS

- A class is a blueprint; an object is a specific instance created from it with new.
- Instance members belong to each object; static members belong to the class as a whole.
- Object variables hold references, so assignment and == compare references, not contents.
- A well-designed class models a single, coherent concept.

7.11 Key Terms, Reflection, and Discussion

Class. Object. Instance. Instance Variable. Field. Instance Method. Static Member. Reference. new keyword. Dot Operator.

- Reflection: For your Application Build, which real-world things become classes? For one of them, what are its instance variables, and would any member be static?
- Discussion: A static member is shared across all objects of a class. When is that exactly what you want, and when would it be a design mistake?

7.12 Activity and Application Build, Step 7

APPLICATION BUILD, STEP 7

Define the first real class of your application, turning one of the real-world things you identified in Step 1 into a class with the fields you listed in Step 2 and at least one method. Create two objects from it in a main method, give them different data, and call their method to confirm each object holds its own independent state.

7.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. The relationship between a class and an object is best described as:

a) They are identical b) Class is a blueprint; object is an instance of it c) Object is a blueprint; class is an instance d) Neither relates to the other

2. The keyword used to create an object from a class is:

a) class b) new c) static d) void

3. A member that belongs to the class as a whole, shared by all objects, is:

a) An instance variable b) A static member c) A local variable d) A parameter

4. A variable of a class type actually holds:

a) The object itself b) A reference to the object c) A copy of the class d) A primitive value

5. A well-designed class should:

> a) Hold many unrelated concerns b) Model a single coherent concept c) Avoid methods d) Be entirely static

7.14 References

- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
- Oracle. The Java tutorials: Classes and objects. Oracle Corporation.
- Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley.

End of Chapter 7. Continue to Chapter 8: Methods and Encapsulation

METHODS AND ENCAPSULATION

Behavior, and the Discipline of Hiding Data

"Encapsulation is a promise: that an object's data can only be changed in ways the object itself permits and understands."

8.0 Chapter Overview

Chapter 7 introduced classes and objects. This chapter examines two closely related ideas in depth: methods, the behavior of objects, including how to call them and pass information to them, and encapsulation, the discipline of hiding an object's data behind controlled access. Encapsulation is one of the defining principles of object-oriented programming, and understanding why it matters, not just how to do it, is essential to writing robust, maintainable object-oriented code.

8.1 Learning Outcomes

- LO 8.1 Explain what methods are and how to call them and pass parameters.
- LO 8.2 Explain encapsulation and why it protects an object's data.
- LO 8.3 Use private fields with public methods to control access to an object's data.

8.2 Introduction

An object's behavior is expressed through its methods: named blocks of code that act on the object's data and can receive information through parameters and return a result. Methods are how objects do things, and how one part of a program asks an object to do something. This chapter examines methods in detail, then turns to encapsulation, the principle of keeping an object's data private and exposing it only through controlled methods, which is what makes objects trustworthy building blocks in a larger program.

Encapsulation is easy to state but profound in effect. By making an object's fields private and providing public methods to access and modify them, a class retains control over its own

data, ensuring it is only ever changed in valid ways. An account object that keeps its balance private and permits changes only through deposit and withdraw methods can guarantee its balance never becomes invalid, a guarantee impossible if any code could set the balance directly. This chapter develops both the mechanics and the reasoning behind this central principle.

8.3 Historical Background

Encapsulation, sometimes called information hiding, was articulated as a principle of good software design in the early 1970s, notably in the work of David Parnas, who argued that modules should hide their internal details behind stable interfaces so that changes inside one module would not ripple through the rest of a system. This idea proved foundational to managing the complexity of large software, and object-oriented languages built it directly into their structure through access control on class members.

Java provides access modifiers, principally private and public, that enforce encapsulation at the language level, letting a class declare which of its members are hidden internal details and which form its public interface. This built-in support means encapsulation is not merely a convention a disciplined programmer may follow, but a structure the language actively supports and the compiler helps enforce, consistent with Java's broader philosophy of steering programmers toward robust, maintainable design.

8.4 Definitions

KEY TERMS

Method: A named block of code, belonging to a class, that performs an action and may take parameters and return a value.

Parameter: A variable in a method's declaration that receives a value passed in when the method is called.

Return Value: The result a method sends back to the code that called it.

Encapsulation: The principle of hiding an object's internal data and exposing it only through controlled methods.

Access Modifier: A keyword, such as private or public, that controls where a class member can be accessed from.

8.5 Core Concepts of Methods and Encapsulation

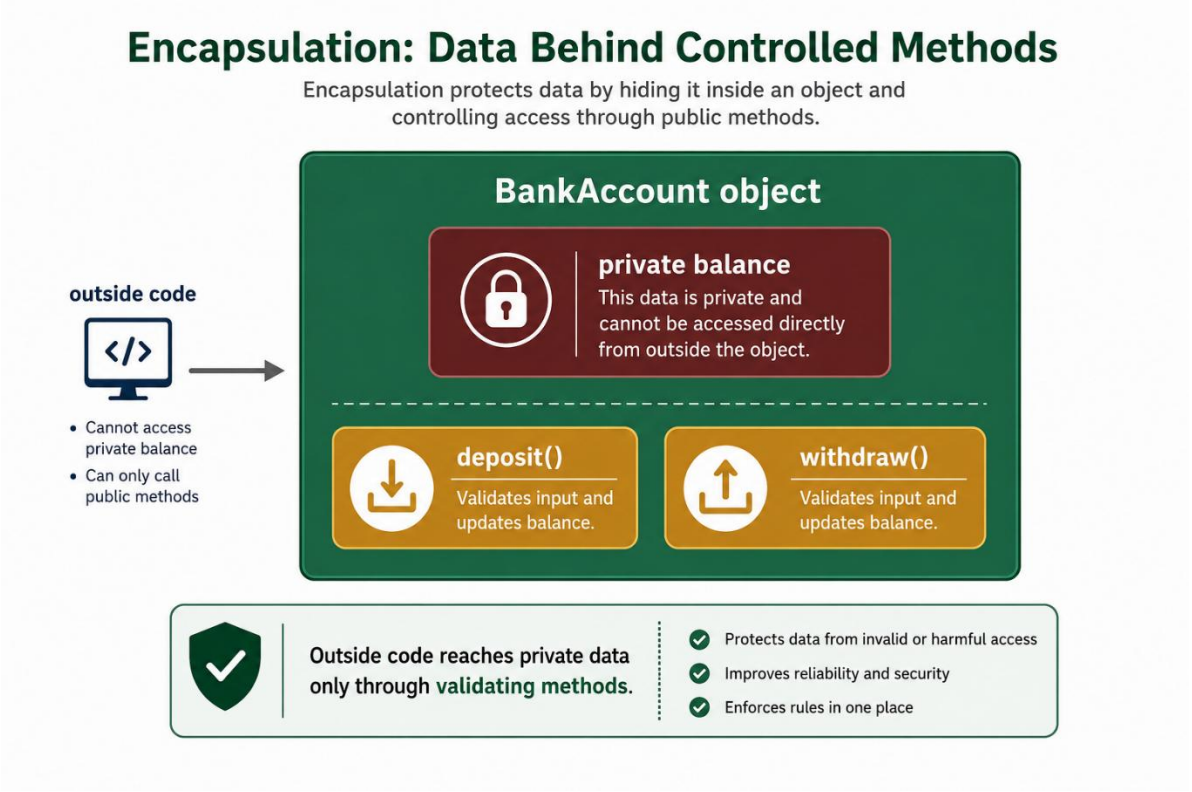

Figure 8.1. Encapsulation: outside code reaches private data only through controlled, validating methods.

8.5.1 Methods, Parameters, and Return Values

A method has a name, an optional list of parameters (the information it receives), a return type (the type of value it sends back, or void if it returns nothing), and a body of statements. To call a method on an object, you use the dot operator and supply values, called arguments, for its parameters. The method runs its body, possibly using the arguments, and if it has a non-void return type, it sends back a value the caller can use. Parameters let the same method operate on different data each time it is called, and return values let a method report a result, together making methods flexible, reusable units of behavior.

8.5.2 Encapsulation in Action

Encapsulation is achieved by making an object's fields private, so no outside code can read or change them directly, and providing public methods that mediate all access. A method that retrieves a field's value is conventionally called a getter, and one that changes it a setter; but the real power comes when these methods enforce rules. The BankAccount class below keeps its balance private and permits changes only through deposit and withdraw methods, each of

which checks that the operation is valid, so the balance can never be set to an invalid value by outside code.

```
public class AccountDemo {
    static class BankAccount {
        private double balance;
        BankAccount(double initial) { this.balance = initial; }
        void deposit(double amount) {
            if (amount > 0) balance = balance + amount;
        }
        boolean withdraw(double amount) {
            if (amount > 0 && amount <= balance) {
                balance = balance - amount;
                return true;
            }
            return false;
        }
        double getBalance() { return balance; }
    }
    public static void main(String[] args) {
        BankAccount acct = new BankAccount(100.0);
        acct.deposit(50.0);
        boolean ok = acct.withdraw(30.0);
        System.out.println("Withdraw succeeded: " + ok);
        System.out.println("Balance: " + acct.getBalance());
    }
}
```

Listing 8.1. Encapsulation with a private field and controlled methods. Verified output: Withdraw succeeded: true / Balance: 120.0

The key to this design is that balance is private: no code outside the class can set it directly. The only ways to change it are deposit, which rejects non-positive amounts, and withdraw, which refuses to overdraw and reports success or failure through its boolean return value. Because every path that modifies the balance is controlled by the class itself, the account can guarantee its balance is always valid. This guarantee is the entire point of encapsulation: the object protects its own data, so the rest of the program cannot corrupt it, even accidentally.

8.5.3 Why Encapsulation Matters

Encapsulation delivers several benefits that grow more valuable as programs grow. It protects an object's data from invalid changes, as the account example shows. It lets a class change its internal implementation without affecting code that uses it, as long as the public methods keep their behavior, since outside code depends only on the interface, not the hidden internals. And it makes a class easier to reason about, since all the ways its data can change are gathered in one place, its own methods, rather than scattered across the whole program. These

benefits are why encapsulation is considered a foundational principle rather than an optional nicety.

8.6 Table 8.1, Access Modifiers

Modifier	Accessible From	Typical Use
private	Only within the same class	Hidden internal data and helper methods
public	Anywhere	The class's intended interface
protected	The class, its subclasses, and package	Members subclasses may need (see Chapter 10)
(default)	Within the same package	Package-internal members

Table 8.1. Common access modifiers and their reach.

8.7 The Getter and Setter Pattern

A common encapsulation pattern provides a getter method to read a private field and, where appropriate, a setter method to change it, with the setter enforcing any validity rules. Not every field needs both: some fields are read-only from outside, exposing only a getter, while others should only ever change through meaningful operations, like deposit, rather than a blunt setter. The discipline is to expose exactly the access each field genuinely needs, and no more, keeping the class in control of its own data. Providing a public setter for every field mechanically, without thought, undermines encapsulation by effectively making the fields public through the back door, so setters should be added deliberately, not reflexively.

8.8 Philippine Context and Case Study

PHILIPPINE CONTEXT

Encapsulation directly supports the data integrity that Philippine financial, banking, and enterprise software depends on absolutely, since these systems cannot tolerate an account balance, inventory count, or transaction record being set to an invalid value by careless code elsewhere in a large system. A developer who encapsulates data properly,

keeping fields private and controlling every change through validating methods, builds the kind of trustworthy components that serious Philippine software systems require, making encapsulation a practical professional necessity rather than an academic ideal.

CASE STUDY

Illustrative composite case.

A Philippine system exposed an account's balance as a public field, and over time various parts of the growing codebase modified it directly, until one poorly written section set a balance to an invalid negative value that should never have been possible, producing errors that were extremely hard to trace because the offending change could have come from anywhere. Refactoring the account to make balance private, changeable only through validating deposit and withdraw methods, both fixed the immediate problem and made future violations impossible. The case illustrates exactly why encapsulation matters: it confines every change to controlled, verifiable paths.

8.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Make fields private and expose them only through methods that enforce any validity rules.
- Best Practice: Change important data only through meaningful operations (like deposit), not blunt setters, where possible.
- Best Practice: Expose exactly the access each field genuinely needs, and no more.
- Common Pitfall: Making fields public, allowing any code to set them to invalid values from anywhere.
- Common Pitfall: Adding a public setter for every field reflexively, undermining encapsulation through the back door.

- Common Pitfall: Scattering the logic that changes an object's data across the program rather than confining it to the class.

8.10 Summary and Key Takeaways

Methods express an object's behavior, receiving information through parameters and reporting results through return values. Encapsulation, the discipline of keeping fields private and controlling access through methods, is a defining principle of object-oriented programming: it protects data from invalid changes, lets internals change without disturbing callers, and gathers all changes to an object's data in one place. Java's access modifiers enforce it at the language level. Chapter 9 now turns to defining full user-defined classes with constructors and packages.

KEY TAKEAWAYS

- Methods express behavior, taking parameters and returning values.
- Encapsulation keeps fields private and controls access through public methods.
- It protects data, hides internals, and gathers all changes to an object's data in one place.
- Java's access modifiers (private, public, protected) enforce encapsulation at the language level.

8.11 Key Terms, Reflection, and Discussion

Method. Parameter. Argument. Return Value. Encapsulation. Information Hiding. Access Modifier. private. public. Getter. Setter.

- Reflection: In your Application Build, which fields must be protected from invalid change, and what methods should control them?

- Discussion: Encapsulation adds methods and indirection rather than letting code touch data directly. Given that cost, why is it still considered worthwhile? When might the overhead not be justified?

8.12 Activity and Application Build, Step 8

APPLICATION BUILD, STEP 8

Encapsulate the class you built in Step 7: make its fields private, and add public methods that control access, ensuring at least one method enforces a validity rule (for example, rejecting an invalid grade, a negative price, or an overdraft). Demonstrate in a main method that the rule is enforced when invalid data is attempted.

8.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. Information passed into a method when it is called is received by its:

a) Return type b) Parameters c) Class name d) Fields

2. Encapsulation is the principle of:

a) Making all fields public b) Hiding data and controlling access through methods c) Avoiding methods d) Using only static members

3. To hide a field from all outside code, declare it:

a) public b) private c) static d) void

4. A key benefit of encapsulation is that it:

a) Makes code run faster b) Protects data from invalid changes and confines them to one place c) Removes the need for methods d) Eliminates classes

5. Adding a public setter for every private field reflexively:

a) Strengthens encapsulation b) Undermines encapsulation through the back door c) Is always required d) Makes fields static

8.14 References

- Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), 1053 to 1058.
- Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley.
- Oracle. The Java tutorials: Controlling access to members of a class. Oracle Corporation.

End of Chapter 8. Continue to Chapter 9: User-Defined Classes, Constructors, and Packages

USER-DEFINED CLASSES, CONSTRUCTORS, AND PACKAGES

Building Complete, Well-Organized Classes

"A constructor is the moment an object comes into being fully formed, with every promise its class makes already true."

9.0 Chapter Overview

This chapter closes Part III by bringing together everything needed to write complete, well-organized classes of your own. You will learn constructors, which initialize new objects; the this reference and the this() constructor call; overloaded methods and constructors; and how packages and access modifiers organize and protect code across a larger program. With this chapter, you have the full toolkit for defining robust classes, the foundation on which the inheritance and polymorphism of Part IV are built.

9.1 Learning Outcomes

- LO 9.1 Create your own classes with attributes, methods, and constructors.
- LO 9.2 Use this reference and the this() constructor call, and create overloaded methods.
- LO 9.3 Organize classes with packages and control access with access modifiers.

9.2 Introduction

The classes in previous chapters were created and then had their fields set one by one, which is tedious and leaves an object in an incomplete state between creation and full initialization. A constructor solves this by initializing a new object's fields at the moment it is created, so the object is fully formed and valid from the instant it exists. This chapter introduces constructors along with the related tools, the this reference, the this() constructor call, method and constructor overloading, and packages, that together let you write complete, professional-quality classes.

These features are what elevate a class from a bare bundle of fields into a well-formed type that guarantees its own validity and integrates cleanly into a larger program. Constructors ensure objects start life correctly; the this reference resolves naming and lets constructors cooperate; overloading provides flexible ways to create and use objects; and packages organize the many classes of a real application. Together, they complete the class-definition toolkit, which this part of the book has been building.

9.3 Historical Background

The constructor, as a special method for initializing objects, has been part of object-oriented languages since their maturation, reflecting a hard-learned lesson: objects that begin life in an incomplete or invalid state are a persistent source of bugs, so the language should provide a guaranteed point at which an object is properly initialized before anyone can use it. Java's constructors run automatically when an object is created with new, ensuring this initialization cannot be accidentally skipped.

Packages, Java's mechanism for organizing classes into named groups, addressed a different problem that emerged as Java programs grew large: the need to organize hundreds or thousands of classes, avoid naming collisions, and control which classes are visible where. The package system, combined with access modifiers, lets a large program be structured into coherent, loosely coupled units, an organizational discipline essential to the large-scale software that Java was designed to support.

9.4 Definitions

KEY TERMS

Constructor: A special method, named after the class, that initializes a new object when it is created with new.

This Reference: A reference, inside a method or constructor, to the current object it is acting on.

this() Constructor Call: A call from one constructor to another constructor of the same class, reducing duplication.

Overloading: Defining several methods or constructors with the same name but different parameter lists.

Package: A named group of related classes, used to organize code and control access.

9.5 Core Concepts of Complete Classes

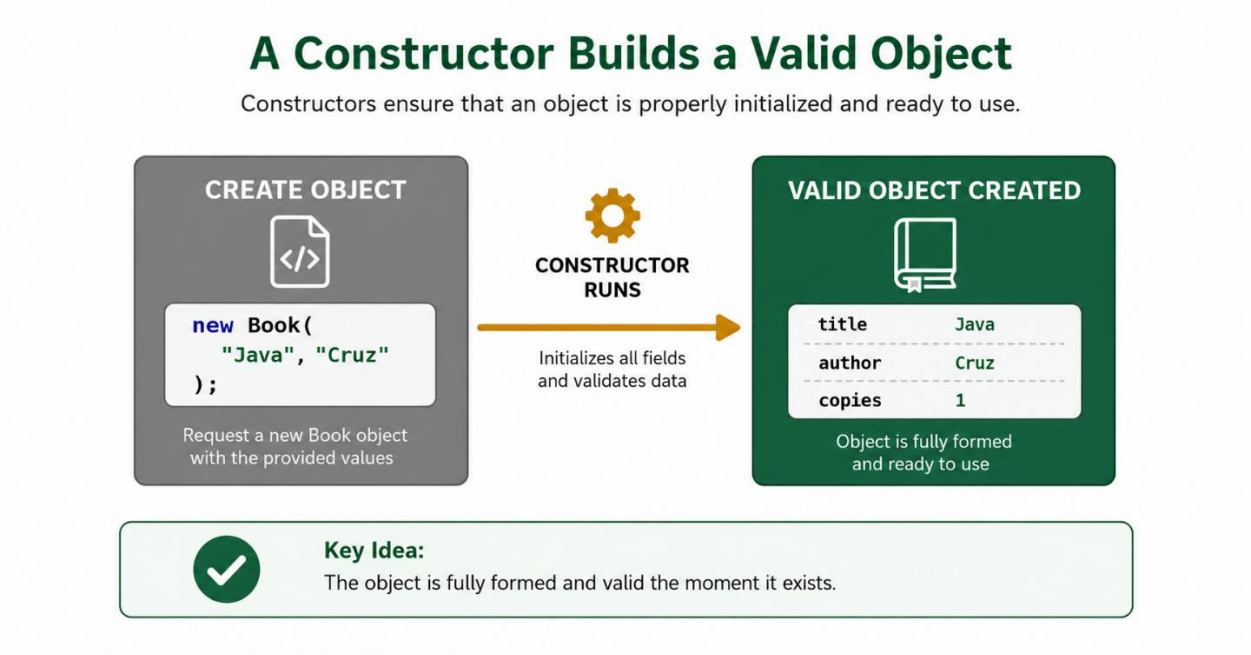

Figure 9.1. A constructor initializes a new object so it is fully formed and valid the moment it exists.

9.5.1 Constructors and the this Reference

A constructor is a special method with the same name as the class and no return type, which runs automatically when an object is created. Its job is to initialize the new object's fields, so the object is valid from the moment it exists. Inside a constructor (or any instance method), the keyword this refers to the object being acted on, which is what lets a constructor parameter share a name with a field: this.name = name assigns the parameter name to the field this.name, distinguishing them clearly. If you write no constructor, Java provides a default one that takes no arguments; but once you need to initialize fields with specific values, you write your own.

9.5.2 Overloading and the this() Call

Overloading means providing several methods, or several constructors, with the same name but different parameter lists, so the caller can use whichever fits their situation. A class might offer one constructor taking full details and another taking only essentials and filling in defaults. To avoid duplicating initialization code, one constructor can call another with this(...), the this() constructor call. The example below shows a Book class with two overloaded constructors, one delegating to the other via this(), demonstrating constructors, this, this(), and overloading together.

```
public class BookDemo {
    static class Book {
        private String title;
        private String author;
        private int copies;
        Book(String title, String author) {
            this(title, author, 1);
        }
        Book(String title, String author, int copies) {
            this.title = title;
            this.author = author;
            this.copies = copies;
        }
        void addCopies(int n) { this.copies += n; }
        String summary() {
            return title + " by " + author + " (" + copies + " copies)";
        }
    }
    public static void main(String[] args) {
        Book b1 = new Book("Java Basics", "Cruz");
        Book b2 = new Book("OOP Design", "Santos", 3);
        b1.addCopies(2);
        System.out.println(b1.summary());
        System.out.println(b2.summary());
    }
}
```

Listing 9.1. Constructors, this, this(), and overloading. Verified output: Java Basics by Cruz (3 copies) / OOP Design by Santos (3 copies)

The two-argument constructor delegates to the three-argument one via this(title, author, 1), supplying a default of one copy, so the initialization logic lives in a single place. The three-argument constructor uses this.title, this.author, and this.copies to assign each parameter to the matching field. Book b1 is created with the two-argument constructor (defaulting to one copy) and then gains two more through addCopies, reaching three; Book b2 is created directly with three copies. This pattern, overloaded constructors with one delegating to another, is idiomatic Java and keeps initialization consistent and free of duplication.

9.5.3 Packages and Access

As a program grows beyond a handful of classes, packages organize them into named groups, declared with a package statement at the top of a file, and made available elsewhere with import, the same mechanism you already used for Scanner. Packages prevent naming collisions and let a program be structured into coherent modules. Access modifiers work together with packages: private members are visible only within their class, public members everywhere, protected members within the package and subclasses, and the default (no modifier) within the package. Choosing access levels deliberately, exposing only what other

code genuinely needs, extends the encapsulation discipline of Chapter 8 from individual classes to the structure of a whole program.

9.6 Table 9.1, Constructor Essentials

Aspect	Detail
Name	Exactly the class name
Return type	None (not even void)
When it runs	Automatically, when new creates an object
Purpose	Initialize the new object's fields to a valid state
Overloading	Multiple constructors with different parameter lists are allowed

Table 9.1. Key facts about constructors.

9.7 Bringing It Together: Well-Formed Classes

With constructors, this, overloading, encapsulation, and packages in hand, you can now write classes that are complete and professional. A well-formed class typically has private fields, one or more constructors that initialize an object to a valid state, methods that express its behavior while enforcing its rules, and a place in a sensibly organized package structure. This is the standard shape of a class in real Java software, and it is the pattern your Application Build classes should now follow. Everything in Part IV, inheritance, polymorphism, abstract classes and interfaces, builds directly on well-formed classes of exactly this kind, so mastering this pattern now is the foundation for what follows.

9.8 Philippine Context and Case Study

> **PHILIPPINE CONTEXT**
>
> Writing complete, well-formed classes with proper constructors and organization is the everyday work of professional Java development in the Philippines, and the package structure and access discipline this chapter introduces are exactly what let the

large teams common in Philippine outsourcing and enterprise software collaborate on shared codebases without stepping on one another. A developer fluent in constructing robust, well-organized classes contributes code that fits cleanly into these large, collaborative systems, a practical competency that this chapter's material directly builds.

CASE STUDY

Illustrative composite case.

A Philippine team found that objects in their system were frequently used before all their fields had been set, because the code created objects and then initialized them field by field, sometimes forgetting a field and leaving objects in an invalid, partially formed state that caused sporadic, hard-to-diagnose errors. Introducing constructors that required the essential data at creation time meant every object was fully valid from the moment it existed, eliminating the whole category of partially initialized-object bugs. The case illustrates precisely why constructors exist: to guarantee that objects begin life correctly, so no code ever encounters a half-formed object.

9.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Provide constructors that require the essential data, so every object is valid from the moment it is created.
- Best Practice: Use this() to delegate from one constructor to another, keeping initialization logic in one place.
- Best Practice: Organize classes into sensible packages and expose only what other code genuinely needs.
- Common Pitfall: Creating objects and then initializing fields one by one, risking partially formed, invalid objects.

- Common Pitfall: Giving a constructor a return type, which turns it into an ordinary method rather than a constructor.
- Common Pitfall: Duplicating initialization logic across overloaded constructors instead of delegating with this().

9.10 Summary and Key Takeaways

Constructors initialize new objects at creation, guaranteeing they begin life valid; the this reference denotes the current object and lets parameters share field names; the this() call lets one constructor delegate to another; overloading provides multiple constructors or methods with different parameter lists; and packages with access modifiers organize and protect a larger program. Together these complete the toolkit for writing well-formed classes. This chapter closes Part III; Part IV now builds on well-formed classes to explore inheritance, polymorphism, and interfaces.

KEY TAKEAWAYS

- A constructor, named after the class with no return type, initializes a new object when new runs.
- this refers to the current object; this() delegates from one constructor to another.
- Overloading provides multiple constructors or methods with different parameter lists.
- Packages and access modifiers organize and protect the classes of a larger program.

9.11 Key Terms, Reflection, and Discussion

Constructor. this Reference. this() Constructor Call. Overloading. Package. import. Access Modifier. Default Constructor. Classpath.

- Reflection: What data must your Application Build's objects require at creation to be valid? What should your constructor's parameters be?
- Discussion: Overloading lets several constructors share a name. How does this improve a class's usability, and when might too many overloaded constructors become confusing?

9.12 Activity and Application Build, Step 9

APPLICATION BUILD, STEP 9

Complete your class into a well-formed type: add one or more constructors that initialize a valid object at creation (using this to assign fields), provide at least two overloaded constructors with one delegating to the other via this(), and confirm every object your program creates is fully valid from the start. Note how you would organize your growing set of classes into packages.

9.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. A constructor's name must be:

a) main b) Exactly the class name c) Any name with void d) The word constructor

2. The keyword this refers to:

a) The class itself b) The current object being acted on c) A static member d) The superclass

3. Calling one constructor from another in the same class uses:

a) super() b) this() c) new d) import

4. Defining several constructors with different parameter lists is called:

a) Overriding b) Overloading c) Encapsulation d) Inheritance

5. Packages are used primarily to:

a) Speed up code b) Organize classes and control access c) Replace constructors d) Store primitive values

9.14 References

- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
- Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley.
- Oracle. The Java tutorials: Providing constructors for your classes; Creating and using packages. Oracle Corporation.

End of Chapter 9. Continue to Chapter 10: Inheritance

PART IV

THE PILLARS OF OBJECT-ORIENTED PROGRAMMING

Part IV explores the three pillars that give object-oriented programming its power: inheritance, by which classes build upon one another; polymorphism, by which one interface serves many behaviors; and abstract classes and interfaces, by which programs are designed around contracts rather than concrete implementations.

INHERITANCE

Building New Classes on Existing Ones

"Inheritance lets a program say what it means directly: a manager is an employee, a circle is a shape, and shares everything that sameness implies."

10.0 Chapter Overview

Part III taught you to build well-formed individual classes. Part IV explores the three pillars that give object-oriented programming its real power, beginning with inheritance. This chapter explains how a new class can be built upon an existing one, inheriting its fields and methods while adding or changing behavior. You will learn superclasses and subclasses, the super keyword, method overriding, and final methods and classes. Inheritance lets related classes share a common structure without duplication and establishes the polymorphism of Chapter 11.

10.1 Learning Outcomes

- LO 10.1 Explain inheritance and define superclasses and subclasses.
- LO 10.2 Use the super keyword to access superclass constructors and members.
- LO 10.3 Override superclass methods and create final methods and classes.

10.2 Introduction

Programs often contain classes that are clearly related: an Employee and a Manager, a Shape and a Circle, an Account and a SavingsAccount. In each pair, the second is a specialized kind of the first, sharing much of its structure while adding or changing some behavior. Inheritance captures exactly this relationship, letting a subclass be built upon a superclass, automatically gaining the superclass's fields and methods while adding its own or modifying inherited ones. This avoids duplicating shared code and expresses the real is-a relationship between the classes directly.

Inheritance is one of the defining features of object-oriented programming, but it must be used thoughtfully. Its proper use is to model a genuine is-a relationship: a Manager truly is an Employee, so Manager inheriting from Employee is sound. Using inheritance merely to reuse code where no genuine is-a relationship exists leads to confusing designs, a caution this chapter emphasizes alongside the mechanics. Used well, inheritance is powerful; used carelessly, it creates tangled hierarchies that are hard to understand.

10.3 Historical Background

Inheritance was among the earliest and most celebrated features of object-oriented languages, present in Simula and central to Smalltalk, because it so directly captured the way people naturally classify things into general categories and specialized subcategories. The ability to define a general class once and derive specialized versions from it promised enormous savings in code and a natural way to model hierarchical relationships, making it a hallmark of the object-oriented approach.

Experience over the following decades tempered this early enthusiasm with hard-won wisdom about the limits of inheritance. Deep or careless inheritance hierarchies can create tight coupling between classes and make systems harder, not easier, to change. This led to a mature consensus, reflected in modern guidance, that inheritance should model genuine is-a relationships and be used judiciously, with composition (building objects from other objects) often preferable for mere code reuse. Java supports this balanced view, providing full inheritance while its idioms encourage its appropriate use.

10.4 Definitions

KEY TERMS

Inheritance: A mechanism by which a class (subclass) is built upon another (superclass), gaining its fields and methods.

Superclass (Parent): The class being inherited from, providing shared fields and methods.

Subclass (Child): The class that inherits from a superclass, adding or modifying behavior.

super Keyword: A reference used to call the superclass's constructor or access its members from a subclass.

> **Method Overriding:** Redefining an inherited method in a subclass to change its behavior.

10.5 Core Concepts of Inheritance

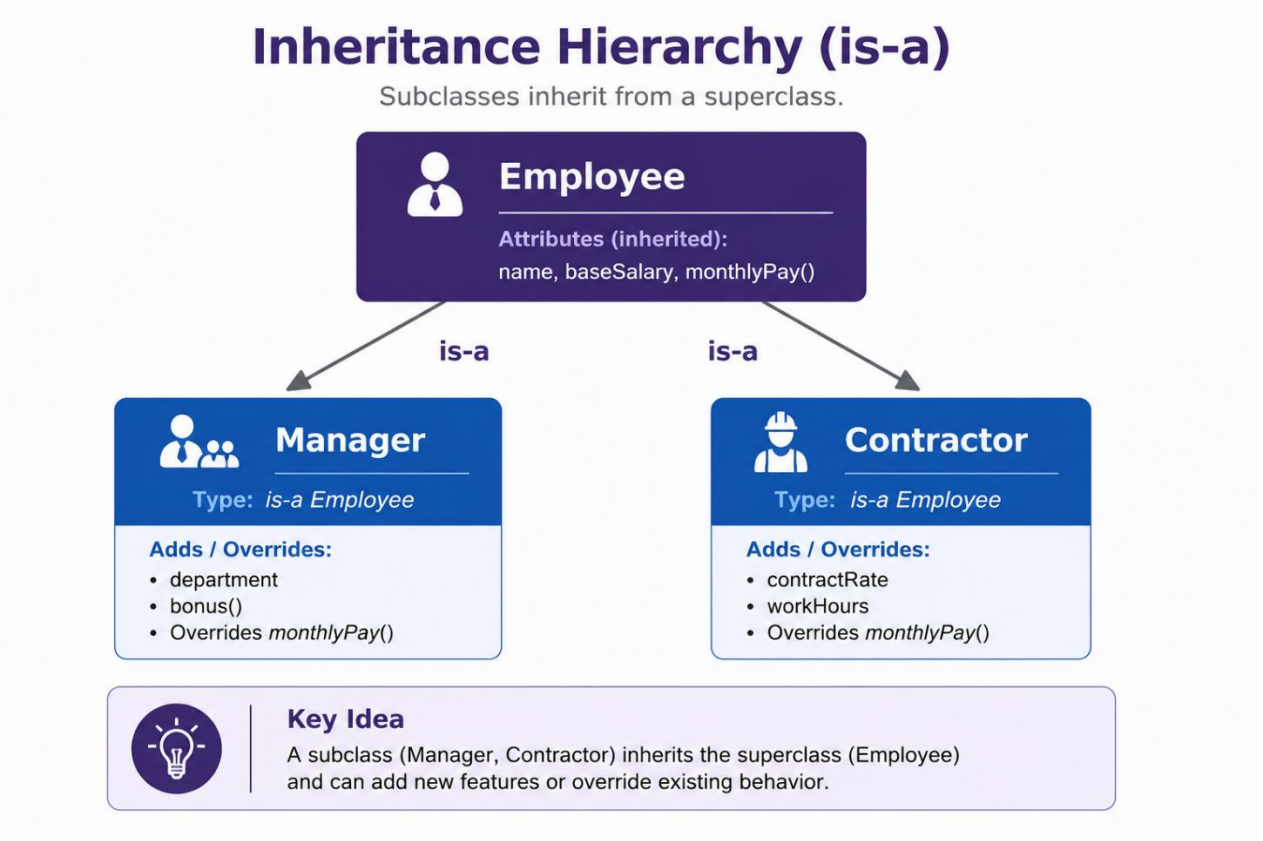

Figure 10.1. An inheritance hierarchy: subclasses inherit a superclass through a genuine is-a relationship.

10.5.1 Superclasses, Subclasses, and extends

A subclass is declared to inherit from a superclass with the keyword extends, as in class Manager extends Employee. The subclass automatically gains all the accessible fields and methods of the superclass, so it need not redeclare them, and it can add its own fields and methods on top. This models the is-a relationship: a Manager is an Employee, and so has everything an Employee has, plus whatever is specific to being a manager. The superclass captures what is common, and each subclass captures what is specific, eliminating the duplication that separate, unrelated classes would require.

10.5.2 The super Keyword and Overriding

A subclass constructor usually needs to initialize the fields it inherited, which are the superclass's responsibility. The super keyword calls the superclass's constructor, as super(name, baseSalary), ensuring the inherited part of the object is initialized before the subclass adds its own. A subclass can also override an inherited method, providing its own version that replaces the superclass's for objects of the subclass; the override must have the same name and parameters, and is conventionally marked with the @Override annotation, which asks the compiler to confirm it genuinely overrides something. The example below shows a Manager subclass overriding monthlyPay to add a bonus.

```
public class InheritDemo {
    static class Employee {
        protected String name;
        protected double baseSalary;
        Employee(String name, double baseSalary) {
            this.name = name;
            this.baseSalary = baseSalary;
        }
        double monthlyPay() { return baseSalary; }
        String describe() { return name + " earns " + monthlyPay(); }
    }
    static class Manager extends Employee {
        private double bonus;
        Manager(String name, double baseSalary, double bonus) {
            super(name, baseSalary);
            this.bonus = bonus;
        }
        @Override
        double monthlyPay() { return baseSalary + bonus; }
    }
    public static void main(String[] args) {
        Employee e = new Employee("Ana", 30000);
        Manager m = new Manager("Ben", 50000, 15000);
        System.out.println(e.describe());
        System.out.println(m.describe());
    }
}
```

Listing 10.1. Inheritance, super, and overriding. Verified output: Ana earns 30000.0 / Ben earns 65000.0

Study what happens with the Manager. Its constructor calls super(name, baseSalary) to let the Employee superclass initialize the inherited name and baseSalary, then sets its own bonus. It overrides monthlyPay to return base salary plus bonus. The revealing detail is describe: it is defined only in Employee and never overridden, yet when called on the Manager, it prints 65000, the manager's overridden pay, because describe calls monthlyPay, and for a Manager object that call reaches the overridden version. This is a first glimpse of polymorphism, the

subject of Chapter 11, and it is one of the most important behaviors in all of object-oriented programming.

10.5.3 final Methods and Classes

Sometimes a class should not be extended, or a method should not be overridden, because doing so would break an assumption the design relies on. The final keyword enforces this: a final method cannot be overridden by a subclass, and a final class cannot be extended at all. Marking a method or class final is a deliberate design statement that its behavior is fixed and must not be changed through inheritance, which can be important for security, correctness, or simply signaling intent. Used judiciously, final communicates and protects design decisions; used excessively, it can make a design rigid, so like inheritance itself, it warrants thoughtful application.

10.6 Table 10.1, Inheritance Vocabulary

Concept	Meaning
extends	Declares that a class inherits from a superclass
super(...)	Calls the superclass constructor from a subclass
Overriding	Replacing an inherited method with a subclass version
@Override	Annotation confirming a method genuinely overrides one
final	Prevents overriding (method) or extending (class)

Table 10.1. Key inheritance concepts.

10.7 Using Inheritance Well

The guiding principle for inheritance is the is-a test: a subclass should genuinely be a kind of its superclass. A SavingsAccount is an Account, so the inheritance is sound; but a class should not inherit from another merely to reuse a convenient method when no real is-a relationship exists, since that produces designs where the class hierarchy misrepresents the actual relationships between concepts. When you want to reuse another class's functionality without a true is-a relationship, composition, having your class hold an instance of the other as

a field, is usually the better choice. Applying the is-a test honestly before reaching for inheritance is what keeps class hierarchies clear and truthful, and it is a discipline worth building from your very first inheritance relationship.

10.8 Philippine Context and Case Study

PHILIPPINE CONTEXT

Inheritance hierarchies structure the large enterprise systems common in the Philippine software and outsourcing industry, where families of related types, different kinds of accounts, employees, products, or transactions, are naturally modeled as subclasses sharing a common superclass. A developer who uses inheritance judiciously, modeling genuine is-a relationships rather than reaching for it merely to reuse code, contributes class hierarchies that remain clear and maintainable as these large systems evolve, an increasingly valued competency as Philippine software projects grow in scale and longevity.

CASE STUDY

Illustrative composite case.

A Philippine team modeling different account types initially made a specialized account inherit from an unrelated class simply to reuse one useful method, producing a hierarchy where the inheritance relationship did not reflect any real is-a relationship and left later developers confused about why the classes were connected. Rewriting the design so the specialized account inherited only from a genuine Account superclass, and obtained the other functionality through composition, made the hierarchy honest and comprehensible. The case illustrates this chapter's core caution: inheritance should express a true is-a relationship, not serve as a shortcut for code reuse.

10.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Apply the is-a test, using inheritance only where a subclass genuinely is a kind of its superclass.
- Best Practice: Call super(...) in a subclass constructor to initialize the inherited part of the object.
- Best Practice: Mark overriding methods with @Override so the compiler confirms they genuinely override.
- Common Pitfall: Using inheritance merely to reuse code, where no genuine is-a relationship exists.
- Common Pitfall: Forgetting to call super(...), leaving the inherited part of an object improperly initialized.
- Common Pitfall: Building deep, tangled inheritance hierarchies that become hard to understand and change.

10.10 Summary and Key Takeaways

Inheritance lets a subclass build upon a superclass with extends, inheriting its fields and methods while adding or overriding behavior, modeling a genuine is-a relationship. The super keyword initializes the inherited part of an object and accesses superclass members; overriding replaces an inherited method; and final prevents overriding or extending. Inheritance should express a true is-a relationship, not serve as a shortcut for reuse. This sets up Chapter 11, where overriding blossoms into full polymorphism.

KEY TAKEAWAYS

- Inheritance (extends) builds a subclass on a superclass, gaining its fields and methods.
- super(...) initializes the inherited part of an object; overriding replaces an inherited method.
- final prevents a method from being overridden or a class from being extended.
- Use inheritance for genuine is-a relationships, not merely to reuse code.

10.11 Key Terms, Reflection, and Discussion

Inheritance. Superclass. Subclass. extends. super. Method Overriding. @Override. final. is-a Relationship. Composition.

- Reflection: Does your Application Build have any genuine is-a relationships that inheritance would model well? What would the superclass and subclasses be?
- Discussion: Modern guidance often favors composition over inheritance for code reuse. Why might inheritance, despite its appeal, be the riskier default?

10.12 Activity and Application Build, Step 10

APPLICATION BUILD, STEP 10

If your application has a genuine is-a relationship, model it with inheritance: create a superclass that captures the commonality and at least one subclass that adds or overrides behavior, using super in the subclass constructor and @Override on any

overridden methods. If no natural is-a relationship exists, explain why, and identify where composition would serve instead.

10.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. The keyword declaring that a class inherits from another is:

a) implements b) extends c) inherits d) super

2. To call the superclass constructor from a subclass, use:

a) this() b) super(...) c) new d) extends

3. Redefining an inherited method in a subclass is called:

a) Overloading b) Overriding c) Encapsulation d) Composition

4. A final method:

a) Cannot be overridden b) Must be overridden c) Is always static d) Cannot be called

5. Inheritance should be used to model:

a) Any code reuse b) A genuine is-a relationship c) Only static members d) Unrelated classes

10.14 References

- Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley.
- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
- Oracle. The Java tutorials: Inheritance. Oracle Corporation.

End of Chapter 10. Continue to Chapter 11: Polymorphism

POLYMORPHISM

One Interface, Many Behaviors

"Polymorphism is how a program can treat many different things the same way, and still have each of them do the right thing."

11.0 Chapter Overview

This chapter examines polymorphism, the second pillar of object-oriented programming and, for many, its most powerful idea. Polymorphism lets a single piece of code work with objects of many different types through a common superclass or interface, with each object responding in its own way. Building directly on the inheritance and overriding of Chapter 10, polymorphism is what lets a program be written in general terms yet behave specifically, and it is the key to flexible, extensible object-oriented design.

11.1 Learning Outcomes

- LO 11.1 Explain polymorphism and how it builds on inheritance and overriding.
- LO 11.2 Treat objects of different subclasses uniformly through a common supertype.
- LO 11.3 Explain how dynamic dispatch selects the correct overridden method at runtime.

11.2 Introduction

The word polymorphism means many forms, and in object-oriented programming, it refers to the ability of a single reference type to refer to objects of many different classes, each responding to the same method call in its own way. If Circle and Rectangle both extend Shape and both override an area method, then a variable of type Shape can hold either, and calling area on it computes the correct area for whichever kind of shape it actually holds. This lets you write code that works uniformly with all shapes, present and future, without knowing or caring which specific kind each one is.

Polymorphism is what transforms inheritance from a mere code-sharing convenience into a source of genuine flexibility. A program can maintain a collection of Shape references, ask each to compute its area, and get the right answer for everyone, even though the collection mixes circles, rectangles, and any other shapes. Crucially, new shape types can be added later without changing the code that processes them, as long as they extend Shape and override area. This extensibility is one of the deepest benefits of object-oriented design, and this chapter shows exactly how it works.

11.3 Historical Background

Polymorphism, particularly the runtime selection of an overridden method based on an object's actual type, was a defining innovation of early object-oriented languages and a major reason the paradigm was considered revolutionary. It allowed programs to be written against general abstractions rather than specific types, a shift that made large systems far more flexible and extensible than the procedural alternative, in which handling a new kind of thing typically meant modifying the code that processed all the existing kinds.

The mechanism that makes polymorphism work, choosing which overridden method to run based on an object's actual runtime type rather than the declared type of the reference, is called dynamic dispatch or late binding, and it became a standard feature of object-oriented language runtimes. Java implements it automatically for overridden instance methods, so polymorphic behavior is the default and requires no special syntax. This automatic, pervasive polymorphism is central to how idiomatic Java programs are structured, favoring general supertypes over specific ones wherever flexibility is wanted.

11.4 Definitions

KEY TERMS

Polymorphism: The ability of a single reference type to refer to objects of many classes, each responding to a method call in its own way.

Supertype Reference: A variable declared as a superclass or interface type that can hold objects of any of its subtypes.

Dynamic Dispatch (Late Binding): Selecting which overridden method to run based on an object's actual runtime type.

> **Upcasting:** Treating a subclass object as its superclass type, which is always safe.

11.5 Core Concepts of Polymorphism

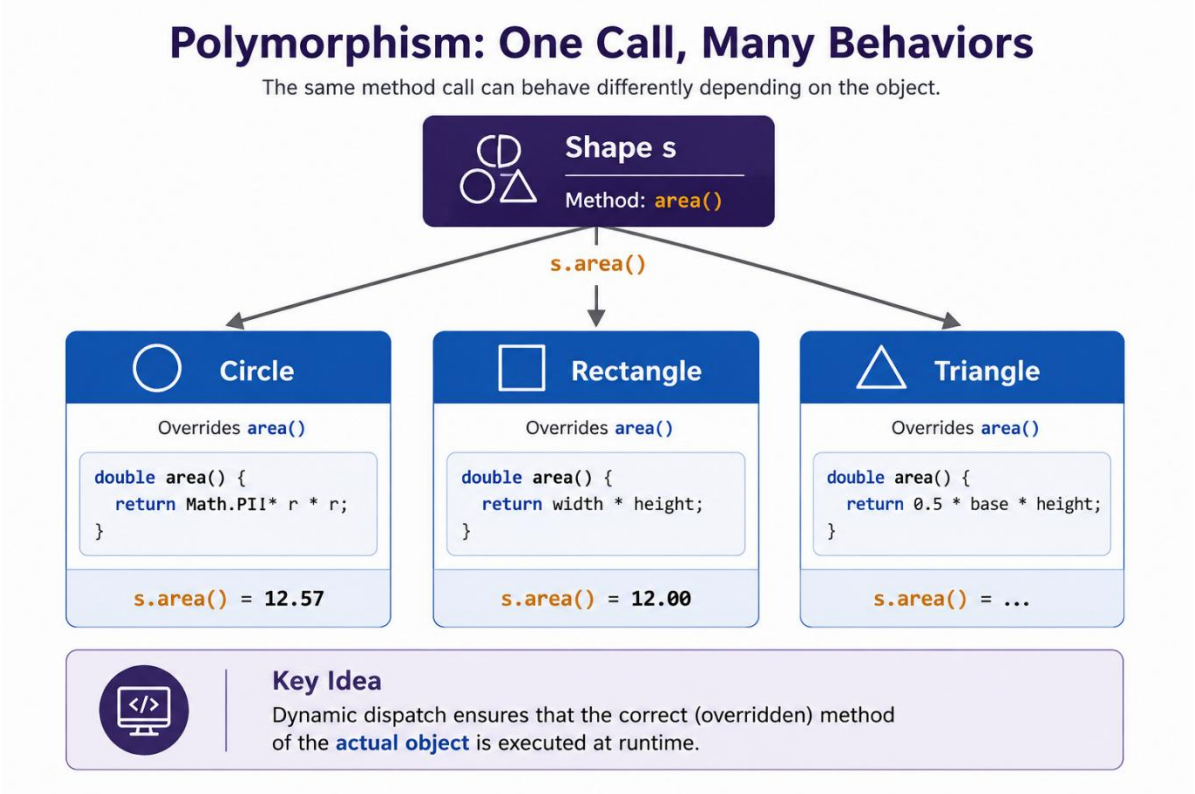

Figure 11.1. Polymorphism: one method call runs each object's own overridden version via dynamic dispatch.

11.5.1 Supertype References Holding Subtype Objects

The foundation of polymorphism is that a variable declared as a superclass type can hold an object of any subclass. A Shape variable can hold a Circle or a Rectangle, because each is a Shape. This is called upcasting, and it is always safe, since a subclass object genuinely is an instance of its superclass. Once held in a supertype reference, the object can be used through the supertype's methods without the code needing to know which specific subclass it is. This lets a single variable, parameter, or array element work with objects of many types uniformly.

11.5.2 Dynamic Dispatch in Action

The magic happens when you call an overridden method through a supertype reference: Java selects the version defined by the object's actual runtime type, not the declared type of the

reference. This is dynamic dispatch. The program below stores a Circle and a Rectangle in an array of Shape, then loops over them calling area on each; despite all being handled as Shape, each call runs the correct overridden area for the actual object.

```
public class PolyDemo {
    static class Shape {
        double area() { return 0.0; }
    }
    static class Circle extends Shape {
        private double r;
        Circle(double r) { this.r = r; }
        @Override double area() { return Math.PI * r * r; }
    }
    static class Rectangle extends Shape {
        private double w, h;
        Rectangle(double w, double h) { this.w = w; this.h = h; }
        @Override double area() { return w * h; }
    }
    public static void main(String[] args) {
        Shape[] shapes = { new Circle(2), new Rectangle(3, 4) };
        for (Shape s : shapes) {
            System.out.printf("Area: %.2f%n", s.area());
        }
    }
}
```

Listing 11.1. Polymorphism via dynamic dispatch. Verified output: Area: 12.57 / Area: 12.00

The loop variable s is declared as Shape, yet s.area() computes 12.57 for the circle (pi times radius squared) and 12.00 for the rectangle (width times height). The same line of code, s.area(), runs different method bodies depending on the actual object, chosen automatically at runtime. This is the essence of polymorphism, and its power is in what it makes possible: the loop needs no knowledge of specific shape types and would work unchanged if a Triangle class were added later, as long as Triangle extends Shape and overrides area. The processing code is written once, in general terms, and accommodates any number of shape types.

11.5.3 Why Polymorphism Matters

Polymorphism's payoff is flexibility and extensibility. Code written against a supertype works with all present and future subtypes without modification, so a system can grow by adding new subclasses rather than editing existing logic, a property that makes object-oriented systems far easier to extend than procedural ones. It also enables clean, general designs: a payroll system can process an array of Employee references, paying each correctly whether it is a salaried employee, a manager, or a contractor, with the differences encapsulated in each

subclass's overridden methods. This combination of uniform handling with type-specific behavior is what makes polymorphism the powerful organizing principle it is.

11.6 Table 11.1, How Polymorphism Works

Piece	Role
Inheritance	Establishes the subtype relationship
Overriding	Gives each subclass its own version of a method
Supertype reference	Holds an object of any subtype uniformly
Dynamic dispatch	Selects the actual object's overridden method at runtime

Table 11.1. The pieces that make polymorphism work.

11.7 Designing for Polymorphism

To take advantage of polymorphism, design code to depend on general supertypes rather than specific subtypes wherever the specific type does not matter. A method that processes shapes should accept a Shape, not a Circle, so it works with every kind of shape. A collection that holds employees should be a collection of Employee, so it can hold any kind. This habit, program to the supertype, is one of the most valuable in object-oriented design, because it is precisely what lets new subtypes slot in without disturbing existing code. Chapter 12's abstract classes and interfaces make this even more powerful by defining supertypes expressly designed to be implemented in many ways.

11.8 Philippine Context and Case Study

PHILIPPINE CONTEXT

Polymorphism underlies the extensible design of the large, evolving software systems built across the Philippine IT industry, where the ability to add new types, new payment methods, new account kinds, new report formats, without rewriting the code that processes them is a direct competitive and maintenance advantage. A developer

who designs against general supertypes, letting polymorphism absorb future variation, builds systems that adapt gracefully to changing requirements, exactly the quality that serious, long-lived Philippine enterprise software demands.

CASE STUDY

Illustrative composite case.

A Philippine system processed different payment methods with a long chain of decisions that checked each payment's type and handled it specifically, so that adding a new payment method meant editing that central chain and risked breaking the existing cases. Redesigning around polymorphism, a common Payment supertype with each method as a subclass overriding a process method, meant the processing code simply called process on each payment and let dynamic dispatch run the right logic, and new payment methods could be added as new subclasses without touching the processing code at all. The case illustrates polymorphism's central benefit: extensibility without modification.

11.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Program to the supertype, accepting and storing general types so new subtypes slot in without changes.
- Best Practice: Let polymorphism replace long type-checking decision chains with overridden methods on each subtype.
- Best Practice: Mark overridden methods with @Override to make the polymorphic design explicit and compiler-checked.
- Common Pitfall: Writing chains of type checks that must be edited whenever a new subtype is added, instead of using polymorphism.

- Common Pitfall: Declaring variables and parameters as specific subtypes when a supertype would allow polymorphic flexibility.
- Common Pitfall: Forgetting that the actual runtime object, not the declared reference type, determines which overridden method runs.

11.10 Summary and Key Takeaways

Polymorphism lets a single supertype reference hold objects of many subtypes, with dynamic dispatch selecting each object's own overridden method at runtime. Built on inheritance and overriding, it lets code written in general terms behave specifically, and lets systems grow by adding subtypes rather than editing existing logic. Programming to the supertype is the design habit that unlocks this flexibility. Chapter 12 now introduces abstract classes and interfaces, supertypes designed expressly for polymorphism.

KEY TAKEAWAYS

- Polymorphism lets one supertype reference hold objects of many subtypes.
- Dynamic dispatch selects the actual object's overridden method at runtime.
- Code written against a supertype works with all present and future subtypes.
- Programming to the supertype is the habit that unlocks polymorphic flexibility.

11.11 Key Terms, Reflection, and Discussion

Polymorphism. Supertype Reference. Dynamic Dispatch. Late Binding. Upcasting. Overriding. Program to the Supertype.

- Reflection: Where could your Application Build process a collection of related objects uniformly through a supertype, letting each behave in its own way?

- Discussion: Polymorphism lets new subtypes be added without changing existing code. How does this change the way a system grows over time compared with a design full of type checks?

11.12 Activity and Application Build, Step 11

APPLICATION BUILD, STEP 11

Demonstrate polymorphism in your application: create an array or list of a supertype (from Step 10's inheritance, or a new one), fill it with objects of different subtypes, and process them uniformly through a single loop that calls an overridden method on each. Confirm that each object behaves according to its actual type, and note how a new subtype could be added without changing the loop.

11.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. Polymorphism allows a single supertype reference to:

a) Hold only one specific type b) Hold objects of many subtypes, each behaving in its own way c) Avoid using methods d) Replace inheritance

2. Selecting the overridden method based on an object's actual runtime type is called:

a) Overloading b) Dynamic dispatch c) Encapsulation d) Casting

3. Treating a Circle object as a Shape reference is:

a) Downcasting b) Upcasting (always safe) c) Overloading d) An error

4. A key benefit of polymorphism is that new subtypes can be added:

a) Only by editing all existing code b) Without changing the code that processes the supertype c) Only if they are static d) Only in the same package

5. Which design habit unlocks polymorphic flexibility?

a) Programming to specific subtypes b) Programming to the supertype c) Avoiding inheritance d) Using only primitive types

11.14 References

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns. Addison-Wesley.
- Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley.
- Oracle. The Java tutorials: Polymorphism. Oracle Corporation.

End of Chapter 11. Continue to Chapter 12: Abstract Classes and Interfaces

ABSTRACT CLASSES AND INTERFACES

Designing Around Contracts

"An interface says what a thing must be able to do, and says nothing about how. That silence is precisely its power."

12.0 Chapter Overview

This chapter closes Part IV with two constructs that make polymorphism even more powerful: abstract classes and interfaces. Both let you define a supertype as a contract, specifying what subtypes must be able to do without fully specifying how, so that many different classes can be treated uniformly through that contract. Understanding when to use an abstract class, when to use an interface, and how both enable clean, flexible designs completes your grasp of the object-oriented pillars and prepares you for the application development of Part V.

12.1 Learning Outcomes

- LO 12.1 Explain abstract classes and abstract methods and when to use them.
- LO 12.2 Explain interfaces and how a class implements them.
- LO 12.3 Choose appropriately between an abstract class and an interface.

12.2 Introduction

Sometimes a supertype exists mainly to define a common contract that its subtypes must fulfill, rather than to provide complete, usable behavior of its own. A Shape has an area, but there is no such thing as a generic shape you would ever create directly; every real shape is a specific kind. An abstract class captures this: it can declare methods that subclasses must implement, without implementing them itself, and cannot be instantiated directly. An interface takes the idea further, defining a pure contract of methods that any class, regardless of its place in the inheritance hierarchy, can promise to fulfill.

These constructs are the tools of contract-based design, one of the most powerful ideas in object-oriented programming. By programming against an abstract class or interface, code depends only on what objects can do, not on their concrete types, achieving maximum flexibility. This chapter explains both constructs, the important differences between them, and the practical question every object-oriented designer faces: when to reach for an abstract class and when for an interface.

12.3 Historical Background

The idea of separating an interface, what operations a type supports, from an implementation, how those operations work, is one of the most durable principles in software design, predating object-oriented programming and central to managing complexity in large systems. Object-oriented languages made this principle concrete through abstract types that specify operations without implementing them, letting programs be built around stable contracts while implementations vary and evolve beneath them.

Java made a notable design choice: unlike some languages that allow a class to inherit from multiple superclasses, Java permits a class to extend only one class, but to implement any number of interfaces. This decision avoided the complications of multiple class inheritance while still allowing a class to fulfill many contracts, and it made interfaces central to Java design. Over time, interfaces gained additional capabilities, but their core role, defining a contract that many unrelated classes can implement, has remained the foundation of flexible Java architecture.

12.4 Definitions

KEY TERMS

Abstract Class: A class that cannot be instantiated directly and may declare abstract methods that subclasses must implement.

Abstract Method: A method declared without a body, which subclasses are required to implement.

Interface: A pure contract of methods that any class can implement, regardless of its class hierarchy.

Implements: The keyword by which a class promises to fulfill an interface's contract.

> **Contract:** The set of operations a type guarantees to provide, separate from how it provides them.

12.5 Core Concepts of Abstract Types

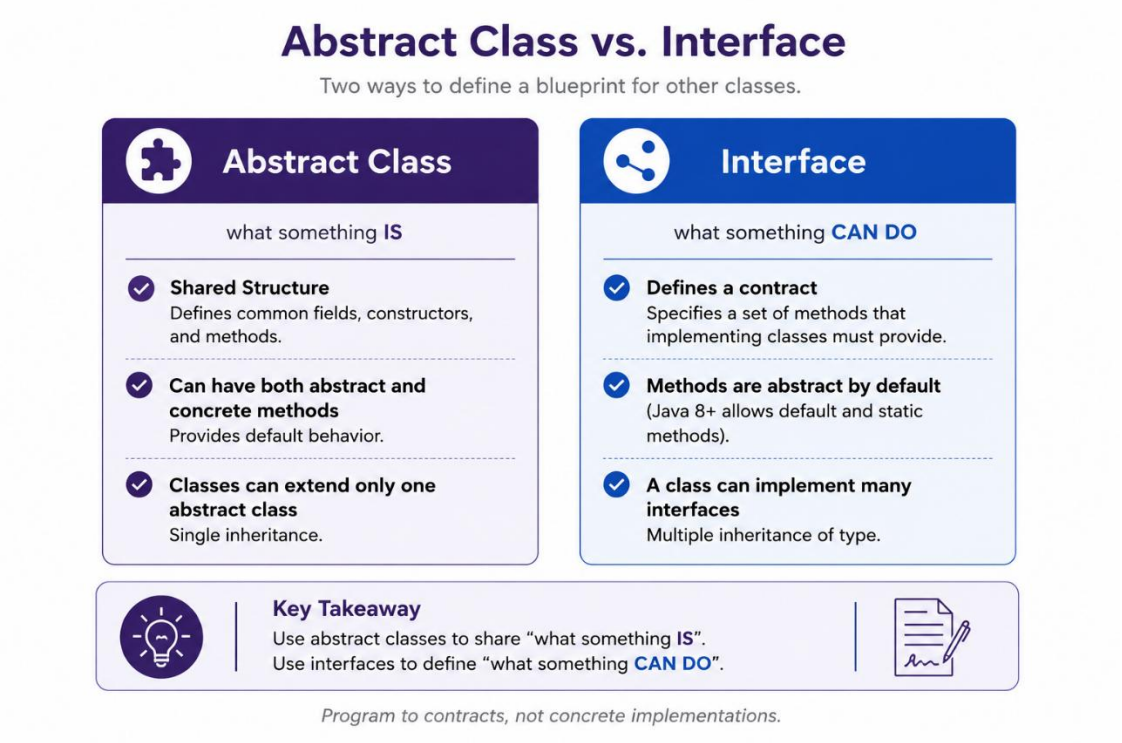

Figure 12.1. An abstract class expresses what something is; an interface expresses what something can do.

12.5.1 Abstract Classes

An abstract class is declared with the abstract keyword and cannot be instantiated directly; you cannot write new on it, because it is incomplete by design. It may contain abstract methods, declared without a body, that every concrete subclass must implement, alongside ordinary methods and fields it does provide. This makes an abstract class ideal when several related classes share substantial common structure and behavior, provided by the abstract class, but each must supply its own version of certain operations, declared abstract. The abstract class thus captures what is shared while requiring subclasses to fill in what differs.

12.5.2 Interfaces

An interface defines a pure contract: a set of method signatures that an implementing class must provide, declared with the interface keyword, and fulfilled by a class using implements.

Unlike a class, an interface traditionally provides no instance fields and specifies only what its methods do, not how. Its great strength is that any class can implement any interface regardless of what it already extends, and a class can implement many interfaces at once, so interfaces cut across the inheritance hierarchy. The example below shows both constructs together: an abstract Account class with an abstract method, and a Payable interface, both fulfilled by a concrete Invoice class.

```
public class AbstractDemo {
    interface Payable {
        double amountDue();
    }
    abstract static class Account {
        protected String owner;
        Account(String owner) { this.owner = owner; }
        abstract String type();
        String label() { return type() + " account for " + owner; }
    }
    static class Invoice extends Account implements Payable {
        private double amount;
        Invoice(String owner, double amount) { super(owner); this.amount = amount; }
        String type() { return "Invoice"; }
        public double amountDue() { return amount; }
    }
    public static void main(String[] args) {
        Invoice inv = new Invoice("Carla", 2500.0);
        System.out.println(inv.label());
        System.out.println("Amount due: " + inv.amountDue());
        Payable p = inv;
        System.out.println("Via interface: " + p.amountDue());
    }
}
```

Listing 12.1. An abstract class and an interface, both fulfilled by one class. Verified output: Invoice account for Carla / Amount due: 2500.0 / Via interface: 2500.0

Invoice both extends the abstract Account (inheriting owner and label, and implementing the required abstract type method) and implements the Payable interface (providing amountDue). Notice label, defined in the abstract class, calls the abstract type method, which the concrete subclass supplies, exactly the polymorphism of Chapter 11 at work through an abstract class. Notice too that the Invoice can be referred to through the Payable interface (Payable p = inv), and calling amountDue through that interface reference works, because Invoice fulfills the Payable contract. This is the power of contract-based design: an object can be viewed and used through any contract it satisfies.

12.5.3 Choosing Between Them

The practical question is when to use each. Use an abstract class when related classes share a common structure and behavior you want to provide once, and there is a genuine is-a relationship, since a class can extend only one abstract class. Use an interface when you want to define a capability that unrelated classes can provide, a Payable, a Comparable, a Drawable, cutting across the class hierarchy, and when a class may need to fulfill several such contracts at once. A rough guide: an abstract class expresses what something is, while an interface expresses what something can do. Many good designs use both, as the example does.

12.6 Table 12.1, Abstract Class vs. Interface

Aspect	Abstract Class	Interface
Expresses	What something is	What can something do
Instance fields	Yes	Traditionally no
Shared method bodies	Yes	Limited
How many per class	Extend only one	Implement many
Relationship	Genuine is-a	A capability, across hierarchies

Table 12.1. Choosing between an abstract class and an interface.

12.7 Programming to Contracts

The deepest lesson of this chapter is to program to contracts, abstract classes and interfaces, rather than to concrete implementations. When code depends only on an interface, it works with any class that implements that interface, present or future, and the implementations can change freely without affecting the code that uses them. This is the fullest expression of the flexibility that polymorphism began: a program built around well-chosen contracts is one whose parts are loosely coupled, independently changeable, and readily extended. This principle underlies much of professional software architecture, and the design patterns and frameworks you will meet in further study rest upon it. For your Application Build, defining a clear contract for a key capability is a step toward exactly this kind of flexible design.

12.8 Philippine Context and Case Study

PHILIPPINE CONTEXT

Interface-based design is fundamental to the modern software frameworks and architectures used throughout the Philippine IT industry, where programming to contracts rather than concrete implementations enables large systems to integrate components, swap implementations, and evolve without cascading changes. A developer fluent in designing with abstract classes and interfaces can build the loosely coupled, extensible systems that professional Philippine software development increasingly demands, making contract-based design a genuinely valued competency in the field.

CASE STUDY

Illustrative composite case.

A Philippine team building a system that needed to send notifications initially wrote code tightly coupled to a single notification method, so that supporting a new channel later required invasive changes throughout the codebase. Redesigning around a Notifier interface, with each channel as a separate implementing class, meant the rest of the system depended only on the interface, and new channels could be added simply by writing new implementations, with no change to the code that sent notifications. The case illustrates this chapter's central lesson: programming to a contract rather than a concrete implementation produces systems that extend cleanly and resist the ripple effects of change.

12.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Program to interfaces and abstract types, so code works with any implementation and tolerates change.

- Best Practice: Use an abstract class for a genuine is-a relationship with shared structure; an interface for a capability across unrelated classes.
- Best Practice: Let a class implement multiple interfaces to fulfill several contracts where appropriate.
- Common Pitfall: Binding code to a concrete class when an interface would allow implementations to vary freely.
- Common Pitfall: Using an abstract class where no genuine is-a relationship exists, when an interface would fit better.
- Common Pitfall: Forgetting that a class can extend only one class but implement many interfaces.

12.10 Summary and Key Takeaways

Abstract classes and interfaces define supertypes as contracts. An abstract class cannot be instantiated and may declare abstract methods subclasses must implement, capturing shared structure for a genuine is-a relationship. An interface defines a pure contract that any class can implement, cutting across the hierarchy, and a class can implement many. Programming to these contracts rather than concrete implementations yields loosely coupled, extensible designs. This chapter closes Part IV; Part V now turns to building robust, complete applications.

KEY TAKEAWAYS

- An abstract class cannot be instantiated and may declare abstract methods subclasses must implement.
- An interface is a pure contract any class can implement, across the inheritance hierarchy.

- A class extends only one class but can implement many interfaces.
- Programming to contracts (abstract types) rather than concrete classes yields flexible, extensible designs.

12.11 Key Terms, Reflection, and Discussion

Abstract Class. Abstract Method. Interface. implements. Contract. Program to an Interface. Loose Coupling.

- Reflection: What capability in your Application Build could be defined as an interface, so that multiple classes might provide it in different ways?
- Discussion: An abstract class expresses what something is; an interface expresses what it can do. Why does Java allow only single class inheritance but multiple interface implementation?

12.12 Activity and Application Build, Step 12

APPLICATION BUILD, STEP 12

Introduce a contract into your application. Define either an abstract class capturing shared structure for a family of related classes, or an interface expressing a capability that one or more of your classes provides, and have at least one class fulfill it. Then write code that uses your objects through the abstract type, demonstrating that it works without depending on the concrete class.

12.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. An abstract class:

a) Can be instantiated directly b) Cannot be instantiated and may declare abstract methods c) Is the same as an interface d) Cannot have any methods

2. A class fulfills an interface's contract using the keyword:

a) extends b) implements c) abstract d) super

3. In Java, a class can:

a) Extend many classes b) Implement only one interface c) Extend one class but implement many interfaces d) Neither extend nor implement

4. A rough guide is that an abstract class expresses what something is, while an interface expresses:

a) What something is not b) What something can do c) How fast something runs d) Where a class is stored

5. Programming to a contract rather than a concrete class produces designs that are:

a) Tightly coupled and rigid b) Loosely coupled and extensible c) Impossible to test d) Always slower

12.14 References

- Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns. Addison-Wesley.
- Oracle. The Java tutorials: Abstract methods and classes; Interfaces. Oracle Corporation.

End of Chapter 12. Continue to Chapter 13: Exception Handling and Robust Code

PART V

APPLICATION DEVELOPMENT

Part V brings everything together into complete, robust applications. It covers the exception handling that makes a program survive the unexpected, and culminates in the integration of every concept the book has taught into a full, working, object-oriented application that solves a real problem.

EXCEPTION HANDLING AND ROBUST CODE

Building Programs That Survive the Unexpected

"The difference between a program and a product is largely what happens when something goes wrong."

13.0 Chapter Overview

Part IV completed the object-oriented pillars. Part V now turns to building complete, robust applications, beginning with exception handling: the mechanism by which a Java program responds to errors and unexpected situations without simply crashing. You will learn what exceptions are, how to catch and handle them with try-catch, how to signal errors by throwing them, and how disciplined exception handling turns a fragile program into a dependable one. This is the foundation of the reliability that real applications require.

13.1 Learning Outcomes

- LO 13.1 Explain what exceptions are and why programs must handle them.
- LO 13.2 Catch and handle exceptions using try-catch.
- LO 13.3 Throw exceptions to signal error conditions, and design for robustness.

13.2 Introduction

Things go wrong in real programs. A user types letters where a number is expected, a file is missing, a network request fails, a calculation attempts to divide by zero. In each case, the normal flow of the program cannot continue, and without a way to respond, the program would simply crash. Java addresses this through exceptions: objects that represent an error or unexpected condition, which the program can catch and handle gracefully rather than letting them terminate the program abruptly. Exception handling is what separates a fragile prototype from software people can actually depend on.

This chapter, and the robustness it teaches, draws together threads from throughout the book. The input handling of Chapter 3 warned that user input is unpredictable; the encapsulation of Chapter 8 protected objects from invalid data; and exception handling now provides the mechanism to respond when, despite these precautions, something goes wrong at runtime. A program that handles exceptions thoughtfully can recover from problems, inform the user clearly, and keep running, which is exactly what real applications must do.

13.3 Historical Background

Before structured exception handling, programmers signaled errors by returning special values, a negative number, a null, a status code, that the caller had to remember to check, an approach that was easy to overlook and cluttered the normal logic with error checks. Structured exception handling, adopted by many languages and built deeply into Java, separated the error-handling path from the normal path, letting the main logic read cleanly while error handling was gathered in dedicated blocks. This was a significant advance in writing reliable software.

Java distinguishes checked exceptions, which the compiler requires a program to handle or declare, from unchecked exceptions, which represent programming errors or conditions a program is not generally expected to recover from. This distinction, though sometimes debated, reflects Java's characteristic emphasis on catching problems early and forcing programmers to acknowledge the ways their code can fail. Whatever one makes of the details, the underlying goal is consistent with the whole language's design: to steer programmers toward writing robust, reliable software rather than code that merely works in the ideal case.

13.4 Definitions

KEY TERMS

Exception: An object representing an error or unexpected condition that disrupts normal program flow.

try-catch: A structure that attempts risky code in a try block and handles any exception in a catch block.

Throwing: Signaling an error condition by creating and raising an exception with the throw keyword.

Handling: Responding to a caught exception so the program can recover or fail gracefully.

Finally: An optional block that runs whether or not an exception occurred, used for cleanup.

13.5 Core Concepts of Exception Handling

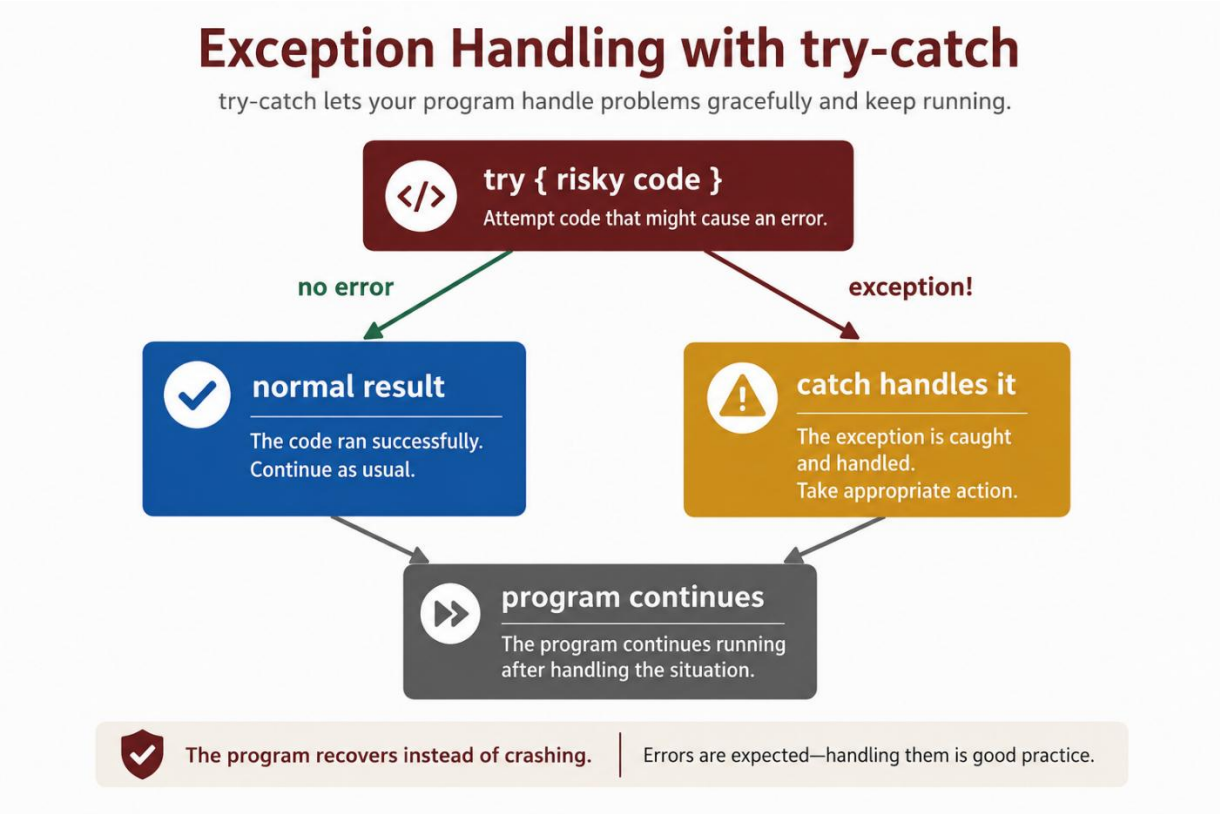

Figure 13.1. The try-catch structure lets a program handle an exception and continue instead of crashing.

13.5.1 Catching Exceptions with try-catch

The try-catch structure is the heart of exception handling. Code that might fail is placed in a try block; if an exception occurs, execution jumps immediately to a matching catch block, which handles the problem, rather than crashing the program. A single try can have multiple catch blocks for different exception types, each handling its own kind of error appropriately. An optional finally block runs afterward whether or not an exception occurred, which is useful for cleanup such as closing a resource. This structure lets the normal logic stay clean while error handling is gathered in dedicated blocks.

13.5.2 Throwing Exceptions

A program can also signal an error itself by throwing an exception with the throw keyword, creating an exception object that describes the problem. This is how a method reports that it was asked to do something invalid, for instance an account rejecting a negative deposit, or a method rejecting a negative age. Throwing an exception passes responsibility for handling the error up to whatever code called the method, which can catch and respond to it. The example below shows both catching and throwing: it safely handles division by zero and invalid number parsing, and throws its own exception for a negative age.

```
public class ExceptionDemo {
    static int safeDivide(int a, int b) {
        try {
            return a / b;
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero.");
            return 0;
        }
    }
    static int parseAge(String text) {
        try {
            int age = Integer.parseInt(text);
            if (age < 0) throw new IllegalArgumentException("Age cannot be negative");
            return age;
        } catch (NumberFormatException e) {
            System.out.println("Not a valid number: " + text);
            return -1;
        } catch (IllegalArgumentException e) {
            System.out.println("Invalid: " + e.getMessage());
            return -1;
        }
    }
    public static void main(String[] args) {
        System.out.println("10 / 2 = " + safeDivide(10, 2));
        System.out.println("5 / 0 = " + safeDivide(5, 0));
        System.out.println("Parsed '25': " + parseAge("25"));
        System.out.println("Parsed 'abc': " + parseAge("abc"));
    }
}
```

Listing 13.1. Catching and throwing exceptions. Verified output: 10 / 2 = 5 / Cannot divide by zero. / 5 / 0 = 0 / Parsed '25': 25 / Not a valid number: abc / Parsed 'abc': -1

Trace the behavior. safeDivide(5, 0) attempts integer division by zero, which throws an ArithmeticException; the catch block reports the problem and returns a safe default rather than letting the program crash. parseAge("abc") calls Integer.parseInt on non-numeric text, which throws a NumberFormatException, caught and handled cleanly. The method also throws its own IllegalArgumentException for a negative age, demonstrating how a method signals an invalid condition. In every case, an error that would otherwise terminate the program is instead

caught and handled, and the program continues running, which is precisely the robustness exception handling provides.

13.5.3 Designing for Robustness

Effective exception handling is more than wrapping code in try-catch; it is a design discipline. Catch exceptions at a level where you can actually do something useful about them, whether recovering, retrying, or informing the user clearly, rather than catching and silently ignoring them, which hides problems and is among the worst habits in programming. Provide informative messages so a failure can be understood. Use exceptions for genuinely exceptional conditions, not for ordinary control flow. And validate input early, as the account and age examples do, so problems are caught close to their source. Together these practices produce programs that fail gracefully and remain dependable under the messy conditions of real use.

13.6 Table 13.1, Exception Handling Constructs

Construct	Purpose
try	Encloses code that might throw an exception
catch	Handles a particular type of exception
throw	Signals an error by raising an exception
finally	Runs cleanup whether or not an exception occurred
getMessage()	Retrieves an exception's descriptive message

Table 13.1. The exception handling constructs.

13.7 Robustness as a Habit

The mindset this chapter instills is to expect the unexpected: to ask, for every operation that could fail, what happens when it does, and to ensure the program responds sensibly. This habit connects to everything you have learned. Encapsulated objects validate their own data; well-designed methods signal invalid requests through exceptions; input handling anticipates malformed input; and try-catch structures let the program recover. A robust application is one built with this defensive mindset throughout, and cultivating it now, as you complete your

Application Build, is what will distinguish the reliable software you write in your career from code that works only until real users encounter it.

13.8 Philippine Context and Case Study

PHILIPPINE CONTEXT

Robustness and graceful error handling are exactly the qualities that distinguish production-grade software in the Philippine industry, especially in the financial, e-commerce, and enterprise systems that cannot afford to crash when a user enters bad data or an external service fails. Employers in the country's software and outsourcing sectors value developers who write defensively, anticipating and handling failure, because such developers produce the dependable systems that real operations require, making exception-handling discipline a concretely marketable professional skill.

CASE STUDY

Illustrative composite case.

A Philippine application worked flawlessly in testing but crashed in the field whenever a user entered unexpected input or a needed file was momentarily unavailable, because the code assumed every operation would succeed. Introducing try-catch around the risky operations, with clear messages and sensible recovery, transformed the program from one that crashed on the first surprise into one that handled problems gracefully and kept running. The case illustrates this chapter's central message: the difference between a demonstration that works and a product that endures lies largely in how carefully the program handles the things that go wrong.

13.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Catch exceptions where you can meaningfully respond, recovering, retrying, or informing the user clearly.

- Best Practice: Validate input early and throw exceptions for genuinely invalid conditions, catching problems near their source.
- Best Practice: Provide informative messages so a failure can be understood and diagnosed.
- Common Pitfall: Catching an exception and silently ignoring it, hiding problems and making bugs nearly impossible to trace.
- Common Pitfall: Using exceptions for ordinary control flow rather than for genuinely exceptional conditions.
- Common Pitfall: Assuming operations will always succeed, leaving the program to crash on the first unexpected situation.

13.10 Summary and Key Takeaways

Exceptions represent errors and unexpected conditions, and exception handling lets a program respond to them gracefully rather than crashing. The try-catch structure attempts risky code and handles any exception; the throw keyword signals an error; and finally handles cleanup. Beyond the mechanics, robustness is a design discipline: catch where you can respond, validate early, message clearly, and expect the unexpected. This defensive mindset is what makes applications dependable. Chapter 14 now brings everything together into a complete application.

KEY TAKEAWAYS

- Exceptions represent errors; handling them lets a program respond gracefully instead of crashing.
- try-catch attempts risky code and handles exceptions; throw signals an error; finally does cleanup.

- Never catch and silently ignore an exception; respond meaningfully or let it propagate.
- Robustness is a design discipline: validate early, message clearly, and expect the unexpected.

13.11 Key Terms, Reflection, and Discussion

Exception. try-catch. throw. Handling. finally. Checked Exception. Unchecked Exception. Robustness. getMessage.

- Reflection: Where in your Application Build could an operation fail, bad input, a missing item, an invalid request? How should the program respond to each?
- Discussion: Silently ignoring a caught exception is considered a serious bad habit. Why is a program that hides its errors often worse than one that fails loudly?

13.12 Activity and Application Build, Step 13

APPLICATION BUILD, STEP 13

Make your application robust. Add try-catch handling around the operations that could fail, especially reading and parsing user input, and throw exceptions from your methods for genuinely invalid conditions. Confirm that your program now responds gracefully, with clear messages, to bad input and invalid requests instead of crashing, and that no catch block silently ignores a problem.

13.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. An exception represents:

a) A normal result b) An error or unexpected condition c) A type of loop d) A class name

2. Code that might fail is placed in which block?

a) catch b) try c) finally d) throw

3. To signal an error condition yourself, you use:

a) catch b) throw c) try d) return

4. A finally block runs:

a) Only when an exception occurs b) Only when no exception occurs c) Whether or not an exception occurred d) Never

5. Catching an exception and silently ignoring it is:

a) Best practice b) A serious bad habit that hides problems c) Required by Java d) The same as throwing it

13.14 References

- Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley.
- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
- Oracle. The Java tutorials: Exceptions. Oracle Corporation.

End of Chapter 13. Continue to Chapter 14: Application Development

APPLICATION DEVELOPMENT

Bringing Every Concept Together

"A finished application is where every separate lesson stops being a topic and becomes a single working thing that solves a real problem."

14.0 Chapter Overview

This final chapter brings together everything the preceding thirteen chapters have taught to develop a complete application. It shows how the language fundamentals, control structures, classes, the object-oriented pillars, and robustness combine into a working program that solves a real problem, and it guides the completion of your cumulative Application Build into a finished piece of software. It also briefly introduces graphical user interfaces and looks ahead to where your Java journey leads next, so that the specific skills you have gained become a foundation for continued growth.

14.1 Learning Outcomes

- LO 14.1 Integrate the book's concepts into a complete, working object-oriented application.
- LO 14.2 Recognize the structure of a graphical user interface application.
- LO 14.3 Design and develop a computing solution using an object-oriented approach.

14.2 Introduction

Every chapter of this book has built toward this point: the ability to design and develop a complete application using an object-oriented approach, the culminating outcome of the entire course. A finished application is where the separate concepts stop being isolated topics and become interdependent parts of a single working program. Its classes encapsulate data and behavior; inheritance and polymorphism organize families of related types; interfaces define

contracts; loops and decisions drive its logic; input and output connect it to its user; and exception handling keeps it robust when things go wrong.

This chapter is candid about what it is and is not. It is not a claim that you now know everything about Java or software development; both are vast, and this course is a foundation, not a completion. What this chapter offers is the integration of what you have learned into the capacity to build real, working, object-oriented software, and an honest map of where to go next. The object-oriented thinking this book has developed is a durable foundation that will serve you across languages and frameworks throughout a career, long after any specific detail here has been superseded.

14.3 Historical Background

Software development matured from the writing of individual programs into the engineering of large systems, a shift that made the organizing principles of object-oriented programming, encapsulation, inheritance, polymorphism, and contract-based design, essential rather than merely elegant. As programs grew from hundreds of lines to millions, the ability to manage complexity through well-designed objects and clear contracts became the difference between systems that could be maintained and extended and those that collapsed under their own weight.

Java itself has continued to evolve since its 1990s origins, adding features that make it more expressive and concise, while its object-oriented core has remained remarkably stable. This stability is a gift to the learner: the fundamentals this book teaches are not a passing fashion but the enduring foundation of the language and, more broadly, of object-oriented programming across many languages. The graphical toolkits, frameworks, and libraries you will meet in further study all rest on exactly the class-and-object thinking you have now learned.

14.4 Definitions

> **KEY TERMS**
>
> **Application:** A complete, working program that solves a real problem, integrating many classes and concepts.

Graphical User Interface (GUI): A visual interface of windows, buttons, and fields, as opposed to a text console.

AWT / Swing: Java's toolkits for building graphical user interface applications.

Collection: An object such as an ArrayList that holds a growable group of objects, beyond the fixed-size array.

14.5 Core Concepts of Application Development

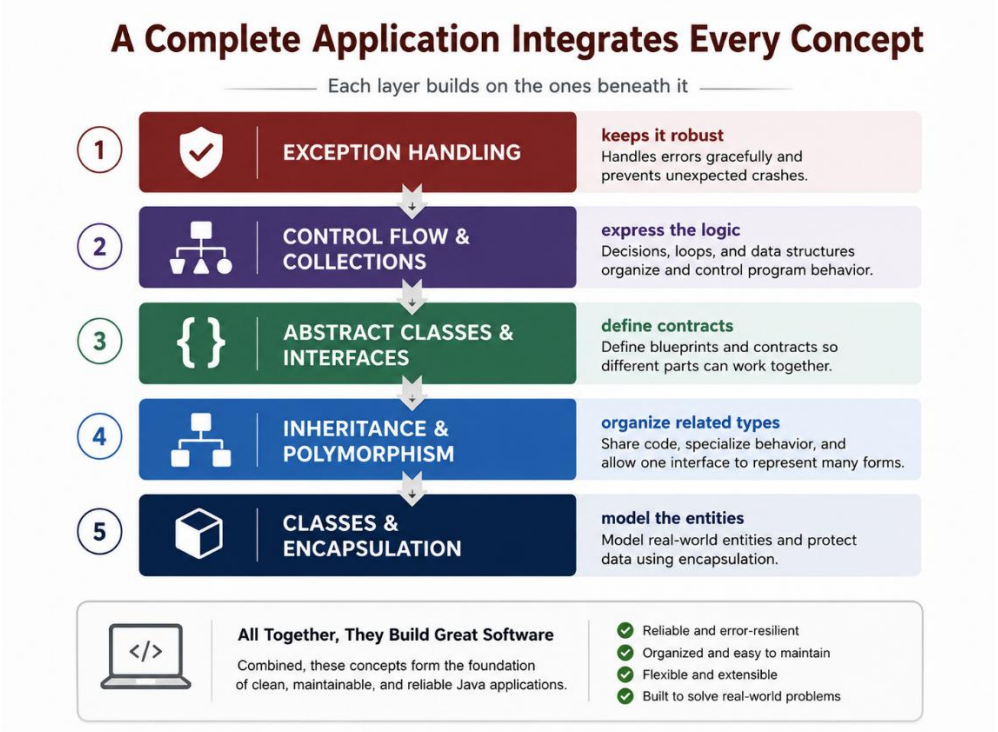

Figure 14.1. A complete application integrates every layer the book has taught into one coherent design.

14.5.1 Integrating the Concepts

A complete application integrates every layer this book has covered. Its classes, designed each around one coherent concept, encapsulate data behind controlled methods. Where genuine is-a relationships exist, inheritance organizes related types, and polymorphism lets them be handled uniformly. Interfaces define the capabilities that cut across the design. Within methods, decisions and loops express the logic, and input and output connect the program to its user. Exception handling makes it robust. These are not separate features bolted together, but interdependent aspects of a single, coherent design, and learning to combine them is the essence of application development.

14.5.2 A Small Complete Application

The program below is a small but complete application, a simple library, that ties together many of the book's concepts: encapsulated classes with private fields and controlled methods, a collection of objects, iteration to search, and methods that enforce rules (a book cannot be borrowed twice). It uses an ArrayList, a growable collection that goes beyond the fixed-size array of Chapter 6, which real applications rely on to hold collections whose size is not known in advance.

```
import java.util.ArrayList;

public class LibraryApp {
    static class Book {
        private String title;
        private boolean borrowed;
        Book(String title) { this.title = title; this.borrowed = false; }
        String getTitle() { return title; }
        boolean isBorrowed() { return borrowed; }
        boolean borrow() {
            if (borrowed) return false;
            borrowed = true;
            return true;
        }
        void giveBack() { borrowed = false; }
    }
    static class Library {
        private ArrayList<Book> books = new ArrayList<>();
        void add(Book b) { books.add(b); }
        Book find(String title) {
            for (Book b : books) {
                if (b.getTitle().equals(title)) return b;
            }
            return null;
        }
        int count() { return books.size(); }
    }
    public static void main(String[] args) {
        Library lib = new Library();
        lib.add(new Book("Java Basics"));
        lib.add(new Book("OOP Design"));
        System.out.println("Books in library: " + lib.count());
        Book b = lib.find("Java Basics");
        if (b != null && b.borrow()) {
            System.out.println("Borrowed: " + b.getTitle());
        }
        System.out.println("Try borrow again: " + b.borrow());
        b.giveBack();
        System.out.println("After return, borrowed? " + b.isBorrowed());
    }
}
```

Listing 14.1. A small complete application. Verified output: Books in library: 2 / Borrowed: Java Basics / Try borrow again: false / After return, borrowed? false

Though small, this application demonstrates the integration that is this chapter's theme. Book encapsulates its state, exposing borrowing only through methods that enforce the rule that a borrowed book cannot be borrowed again, seen when the second borrow attempt returns false. Library holds a collection of books and provides methods to add, find, and count them, iterating to search. The main method uses these classes together to model a real workflow. Scaling this same pattern up, more classes, richer behavior, exception handling, and perhaps a graphical interface, is how full applications are built, and it is exactly what your Application Build has been practicing.

14.5.3 Graphical User Interfaces

The applications in this book have used the text console, but many real applications present a graphical user interface (GUI) of windows, buttons, and fields. Java provides toolkits, the older AWT and the more capable Swing, for building these, and the syllabus's mention of the Abstract Windowing Toolkit points toward this. A GUI application is still built on exactly the object-oriented foundations you have learned: its windows and controls are objects, its responses to user actions are methods, and its design uses classes, inheritance, and interfaces throughout. Graphical programming is a substantial topic for further study, but it introduces no new object-oriented principles, only new libraries of classes to apply the principles you already know.

14.6 Table 14.1, Concepts in a Complete Application

Concept	Contribution	Covered In
Classes and encapsulation	Model the application's entities safely	Chapters 7, 8, 9
Inheritance and polymorphism	Organize and handle related types	Chapters 10, 11
Abstract classes and interfaces	Define contracts and enable flexibility	Chapter 12
Control flow and collections	Express the application's logic	Chapters 4, 5, 6
Exception handling	Keep the application robust	Chapter 13

Table 14.1. How the book's concepts contribute to a complete application.

14.7 Where Your Java Journey Leads Next

Completing this book is a foundation, not an endpoint. From here, natural next steps include the Java Collections Framework (richer data structures beyond arrays and ArrayList), file and database access (storing data permanently), graphical and web interfaces (Swing, JavaFX, and web frameworks), and the design patterns that capture proven solutions to recurring design problems. Beyond Java specifically, the object-oriented thinking you have built transfers directly to many other languages, since the same principles of encapsulation, inheritance, polymorphism, and contract-based design recur throughout modern programming. The specific tools will keep changing across your career; the object-oriented foundation this book has given you will remain the durable basis on which you build. Your completed Application Build is evidence that you can now design and develop real object-oriented software, exactly the outcome this course set out to achieve.

14.8 Philippine Context and Case Study

PHILIPPINE CONTEXT

The ability to design and develop a complete object-oriented application is precisely the competency that makes a computer science graduate employable in the Philippine software industry, from the large outsourcing and enterprise sectors to the country's growing startup and financial technology scene. A graduate who has built a complete application, integrating classes, the object-oriented pillars, and robust error handling into working software, carries demonstrable evidence of exactly the practical capability that Philippine employers seek, which is why this course culminates in a finished application rather than isolated exercises.

CASE STUDY: THE COMPLETE APPLICATION BUILD

Illustrative synthesis of the cumulative project.

A student began this course by choosing to build a simple library management application. Across the chapters, the project grew: Book and Member became encapsulated classes (Chapters 7 to 9); a common LibraryItem superclass with Book

and Magazine subclasses introduced inheritance and polymorphism (Chapters 10, 11); a Borrowable interface defined the borrowing contract (Chapter 12); loops and decisions drove the borrowing logic (Chapters 4, 5); and try-catch handling made the program robust against invalid input (Chapter 13). The student's final deliverable, a complete, working, object-oriented application, is exactly the integrated capability this book set out to build, from the first Hello, world! to a program that solves a real problem.

14.9 Best Practices, Current Issues, and Common Pitfalls

- Best Practice: Build an application as an integrated, coherent design, not a collection of features bolted together.
- Best Practice: Apply object-oriented principles throughout, letting classes, encapsulation, and polymorphism shape the whole design.
- Best Practice: Treat completing this book as a foundation, and keep building on it through collections, persistence, interfaces, and patterns.
- Common Pitfall: Reverting to procedural, all-in-one-method code when building a larger program, abandoning object-oriented structure.
- Common Pitfall: Neglecting robustness in a larger application, so it works in demonstration but fails in real use.
- Common Pitfall: Treating the end of this course as the end of learning, rather than the foundation it is meant to be.

14.10 Summary and Key Takeaways

A complete application integrates every concept this book has taught, encapsulated classes, inheritance and polymorphism, interfaces, control flow, collections, and exception handling, into a single coherent program that solves a real problem. Graphical interfaces build on the same object-oriented foundations using additional libraries. Completing this book is a foundation for continued growth in collections, persistence, interfaces, patterns, and beyond, and the object-oriented thinking it has built transfers across the whole of modern programming. This completes the main text.

KEY TAKEAWAYS

- A complete application integrates the book's concepts into one coherent, working program.
- GUI applications apply the same object-oriented foundations through additional toolkits (AWT, Swing).
- This book is a foundation; next steps include collections, persistence, interfaces, and design patterns.
- Object-oriented thinking transfers across languages and outlasts any specific tool or detail.

14.11 Key Terms, Reflection, and Discussion

Application. Graphical User Interface. AWT. Swing. Collection. ArrayList. Integration. Object-Oriented Design.

- Reflection: Looking back across all fourteen chapters, which single object-oriented concept do you expect to rely on most as you continue programming?

- Discussion: The fundamentals in this book outlast specific tools and languages. How will you keep learning as Java, its frameworks, and the wider field continue to evolve?

14.12 Activity and Application Build, Step 14 (Capstone)

APPLICATION BUILD, STEP 14: FINAL COMPILATION

Integrate: Bring together the classes and features from Steps 1 through 13 into a single, complete, working application.

Apply: Ensure it demonstrates encapsulation, at least one inheritance or interface relationship, polymorphism, control flow, a collection of objects, and exception handling.

Verify: Test it with both valid and invalid input, confirming it behaves correctly and handles errors gracefully.

Present: Prepare a short walkthrough of your application, explaining how each object-oriented concept from the book appears in your design.

14.13 Chapter Self-Check

CHAPTER SELF-CHECK

1. A complete application is best understood as:

a) A collection of unrelated features b) An integrated, coherent design combining many concepts c) A single method d) A list of classes with no interaction

2. Java's toolkits for building graphical user interfaces include:

a) Scanner and BufferedReader b) AWT and Swing c) try and catch d) int and double

3. An ArrayList differs from an array in that it:

a) Cannot hold objects b) Can grow as needed c) Has no methods d) Is always faster

4. A GUI application introduces:

a) Entirely new object-oriented principles b) New libraries applying the same object-oriented principles c) No use of classes d) Only procedural code

5. Completing this book is best regarded as:

a) The end of learning b) A foundation for continued growth c) A reason to stop practicing d) Sufficient to master all of software development

14.14 References

- Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley.
- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns. Addison-Wesley.
- Oracle. The Java tutorials: Creating a GUI with Swing. Oracle Corporation.

End of Chapter 14. This completes the main text.

List of Abbreviations

Acronym	Meaning
API	Application Programming Interface
AWT	Abstract Window Toolkit
CO	Course Outcome
GUI	Graphical User Interface
IDE	Integrated Development Environment
JDK	Java Development Kit
JRE	Java Runtime Environment
JIT	Just-In-Time Compiler
JVM	Java Virtual Machine
LO	Learning Outcome
OOP	Object-Oriented Programming

List of Figures

List of Tables

Glossary

The following terms are introduced and defined throughout the fourteen chapters of this book. Each entry lists the chapter in which the term is first defined; refer to that chapter's Definitions section for full context.

Java *(Ch. 1)* A general-purpose, object-oriented programming language that compiles to platform-independent bytecode.

Bytecode *(Ch. 1)* The intermediate, platform-independent form a Java program is compiled to, run by the JVM.

Object-Oriented Programming *(Ch. 1)* A paradigm that organizes a program around objects, which bundle data with the behavior that acts on it.

Class *(Ch. 1)* A blueprint that defines the data (fields) and behavior (methods) of a type of object.

Object *(Ch. 1)* A specific instance of a class, created at runtime, holding its own copy of the instance data.

Primitive Data Type *(Ch. 2)* One of Java's basic built-in types, such as int, double, char, and boolean.

Variable *(Ch. 2)* A named storage location that holds a value of a declared type.

Operator *(Ch. 2)* A symbol that performs an operation on values, such as + for addition or > for comparison.

Scanner *(Ch. 3)* A Java class that reads and converts input from a source such as the keyboard.

Parsing *(Ch. 3)* Converting text into another type, such as turning the string 42 into the integer 42.

Decision Control Structure *(Ch. 4)* A construct that chooses which statements to execute based on a condition.

Condition *(Ch. 4)* A boolean expression, evaluating to true or false, that a decision structure tests.

Loop *(Ch. 5)* A control structure that repeats a block of statements while a condition holds.

Infinite Loop *(Ch. 5)* A loop whose condition never becomes false, so it never terminates.

Array *(Ch. 6)* A fixed-size, ordered collection of values of the same type, stored under a single name.

Index *(Ch. 6)* The position of an element in an array, starting at 0 for the first element.

Instance Variable *(Ch. 7)* A variable that belongs to each object individually, holding that object's own state.

Static Member *(Ch. 7)* A variable or method that belongs to the class as a whole, shared across all objects.

Method *(Ch. 8)* A named block of code, belonging to a class, that performs an action and may take parameters and return a value.

Encapsulation *(Ch. 8)* The principle of hiding an object's internal data and exposing it only through controlled methods.

Access Modifier *(Ch. 8)* A keyword such as private or public that controls where a class member can be accessed from.

Constructor *(Ch. 9)* A special method, named after the class, that initializes a new object when it is created with new.

this Reference *(Ch. 9)* A reference, inside a method or constructor, to the current object it is acting on.

Overloading *(Ch. 9)* Defining several methods or constructors with the same name but different parameter lists.

Package *(Ch. 9)* A named group of related classes, used to organize code and control access.

Inheritance *(Ch. 10)* A mechanism by which a subclass is built upon a superclass, gaining its fields and methods.

Method Overriding *(Ch. 10)* Redefining an inherited method in a subclass to change its behavior.

super Keyword *(Ch. 10)* A reference used to call the superclass's constructor or access its members from a subclass.

Polymorphism *(Ch. 11)* The ability of a single reference type to refer to objects of many classes, each responding in its own way.

Dynamic Dispatch *(Ch. 11)* Selecting which overridden method to run based on an object's actual runtime type.

Abstract Class *(Ch. 12)* A class that cannot be instantiated directly and may declare abstract methods subclasses must implement.

Interface *(Ch. 12)* A pure contract of methods that any class can implement, regardless of its class hierarchy.

Exception *(Ch. 13)* An object representing an error or unexpected condition that disrupts normal program flow.

try-catch *(Ch. 13)* A structure that attempts risky code in a try block and handles any exception in a catch block.

Collection *(Ch. 14)* An object such as an ArrayList that holds a growable group of objects, beyond the fixed-size array.

Appendix A: The Complete Application Build Worksheet Packet

This appendix consolidates the fourteen steps of the cumulative Application Build into a single running worksheet. Distributed at the start of the term, it lets a student carry one project from first idea to finished application, completing one step as each chapter is studied. Each step below corresponds to the Application Build activity at the end of the matching chapter.

Step 1 (Ch. 1): Choose a realistic small application and describe what it does; list the real-world things it deals with, which become your objects.

Step 2 (Ch. 2): For each object, list the data it must hold and assign each an appropriate primitive type or String; this becomes your fields.

Step 3 (Ch. 3): Write an interactive program that uses Scanner to gather your application's initial data and prints it back in a formatted confirmation.

Step 4 (Ch. 4): Identify the decisions your application must make and implement at least one real decision rule as an if-else or switch structure.

Step 5 (Ch. 5): Add a loop (for example, a menu that repeats until exit, or a loop over records), choosing the loop type deliberately and confirming it terminates.

Step 6 (Ch. 6): Add an array holding a collection of related values and a loop that processes it, using arrayName.length so the code adapts to any size.

Step 7 (Ch. 7): Define your first real class with the fields from Step 2 and at least one method; create two objects and confirm each holds its own state.

Step 8 (Ch. 8): Encapsulate the class: make its fields private and add methods that control access, with at least one enforcing a validity rule.

Step 9 (Ch. 9): Add constructors (at least two overloaded, one delegating via this()) so every object is fully valid the moment it is created.

Step 10 (Ch. 10): If a genuine is-a relationship exists, model it with inheritance: a superclass and a subclass that overrides behavior using super and @Override.

Step 11 (Ch. 11): Demonstrate polymorphism: process a collection of a supertype through one loop, letting each object's overridden method run.

Step 12 (Ch. 12): Introduce a contract: define an abstract class or interface and have a class fulfill it; use your objects through the abstract type.

Step 13 (Ch. 13): Make the application robust: add try-catch around operations that can fail and throw exceptions for invalid conditions.

Step 14 (Ch. 14): Integrate all steps into a single complete application demonstrating encapsulation, inheritance or interfaces, polymorphism, control flow, a collection, and exception handling; test with valid and invalid input; prepare a walkthrough.

Appendix B: Standards, Conventions, and Philippine Context Quick Reference

This appendix consolidates the conventions, standards, and Philippine-context notes referenced throughout the book into a single quick reference.

Topic	Convention or Standard	Relevance
Class naming	PascalCase (e.g. BankAccount)	Standard Java convention for class names
Variable/method naming	camelCase (e.g. monthlyPay)	Standard Java convention for variables and methods
Constant naming	UPPER_SNAKE_CASE	Standard Java convention for constants
File naming	Matches the public class name	Required by the Java compiler
Encapsulation	private fields, public methods	Protects data integrity in enterprise systems
Override marking	@Override annotation	Compiler-checked confirmation of overriding
Employability	Java OOP competency	Among the most in-demand skills in Philippine IT and BPO
Robustness	Exception handling discipline	Distinguishes production software in Philippine industry

Table B.1. Java standards, conventions, and Philippine context quick reference.

Appendix C: Index of Key Concepts and Course Outcome Alignment

This appendix indexes the major concepts introduced throughout the book, and maps each of the book's five Parts to the three course outcomes and the topic sequence of the SDF104 syllabus.

Concept	Defined In
Object-oriented vs. procedural	Chapter 1
Primitive types and operators	Chapter 2
Input, output, and parsing	Chapter 3
Decision control structures	Chapter 4
Loops and repetition	Chapter 5
Arrays and multidimensional arrays	Chapter 6
Classes, objects, instance vs. static	Chapter 7
Methods and encapsulation	Chapter 8
Constructors, this, overloading, packages	Chapter 9
Inheritance, super, overriding, final	Chapter 10
Polymorphism and dynamic dispatch	Chapter 11
Abstract classes and interfaces	Chapter 12
Exception handling and robustness	Chapter 13
Application development and collections	Chapter 14

Table C.1. Index of key concepts and the chapter in which each is defined.

Course Outcome	Description	Primary Parts/Chapters
CO1	Understand and design object-oriented concepts and object-oriented design	Parts I, III (Ch. 1, 7 to 9)
CO2	Apply standards and principles to write readable object-oriented code	Parts II, III, IV (Ch. 2 to 6, 8, 10 to 12)
CO3	Design and develop a computing solution using an object-oriented approach	Parts IV, V (Ch. 10 to 14)

Table C.2. Alignment of the book's Parts with the three course outcomes and syllabus topics.

Appendix D: Suggested Course Calendar

The following 14-week calendar maps one chapter to one instructional week, suitable for a standard one-semester Java object-oriented programming course. Institutions may instead pace chapters per Part, aligning the five Parts with the midterm and final examination points indicated in the course learning plan.

Week	Chapter	Milestone
Week 1	Chapter 1: Introduction to Java and Object-Oriented Thinking	Application Build Step 1 due
Week 2	Chapter 2: Data Types, Variables, and Operators	Application Build Step 2 due
Week 3	Chapter 3: Getting Input and Producing Output	Application Build Step 3 due
Week 4	Chapter 4: Decision Control Structures	Application Build Step 4 due
Week 5	Chapter 5: Loops and Repetition	Application Build Step 5 due
Week 6	Chapter 6: Arrays	Application Build Step 6 due
Week 7	Chapter 7: Objects and Classes	Application Build Step 7 due
Week 8	Chapter 8: Methods and Encapsulation	Application Build Step 8 due
Week 9	Chapter 9: User-Defined Classes, Constructors, and Packages	Application Build Step 9 due
Week 10	Chapter 10: Inheritance	Application Build Step 10 due
Week 11	Chapter 11: Polymorphism	Application Build Step 11 due
Week 12	Chapter 12: Abstract Classes and Interfaces	Application Build Step 12 due
Week 13	Chapter 13: Exception Handling and Robust Code	Application Build Step 13 due
Week 14	Chapter 14: Application Development	Application Build Step 14 due

Table D.1. Suggested 14-week course calendar.

Suggested Further Reading

Students seeking deeper treatment of specific topics covered in this book may consult the following foundational texts, each cited within the relevant chapter:

Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson. A comprehensive introduction to Java covering the full language.

Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley. Best practices for writing robust, idiomatic Java, valuable after the fundamentals.

Schildt, H. (2021). Java: The complete reference (12th ed.). McGraw-Hill. A thorough reference to the Java language and libraries.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns. Addison-Wesley. The classic catalog of object-oriented design patterns for further study.

Oracle. The Java tutorials. Oracle Corporation. The official, freely available tutorials covering every topic in this book and beyond.

Notes on Sources: The references listed in each chapter are foundational works in Java and object-oriented programming. Before final publication, all citations will undergo a full accuracy audit against APA 7th edition standards, as noted in the Preface.

About the Author

Mr. Tolentino is an information technology educator and academic administrator with extensive experience in teaching computer programming, software development, and other computing-related professional courses. He has served as a faculty member at Pangasinan State University, where he has handled subjects such as Object-Oriented Programming and other courses involving the application of programming principles, problem-solving techniques, and software design. He has also served as College Dean at the university's Lingayen Campus, contributing to the administration, academic development, and implementation of programs within the field of information and computing technology.

His academic and professional interests include object-oriented programming, Java application development, mobile application development, software engineering, educational technology, and the effective integration of digital tools into teaching and learning. He is the author of *Object-Oriented Programming with Java: A Practical Guide to Software Design and Application Development*, an instructional resource designed to help students understand essential programming concepts through structured explanations, practical examples, and application-based activities. He has also participated in collaborative scholarly and technology-related projects, including work involving Android game development.

As an educator, administrator, curriculum contributor, and author, Mr. Tolentino remains committed to improving the quality of information technology education through practical, outcomes-based, and learner-centered instructional approaches. His teaching and academic work emphasize the connection between programming theory and real-world software development, enabling students to strengthen their technical knowledge, logical reasoning, creativity, and problem-solving abilities. Through his continuing involvement in instruction, academic leadership, and learning-resource development, he contributes to the preparation of competent, ethical, innovative, and industry-ready computing professionals.